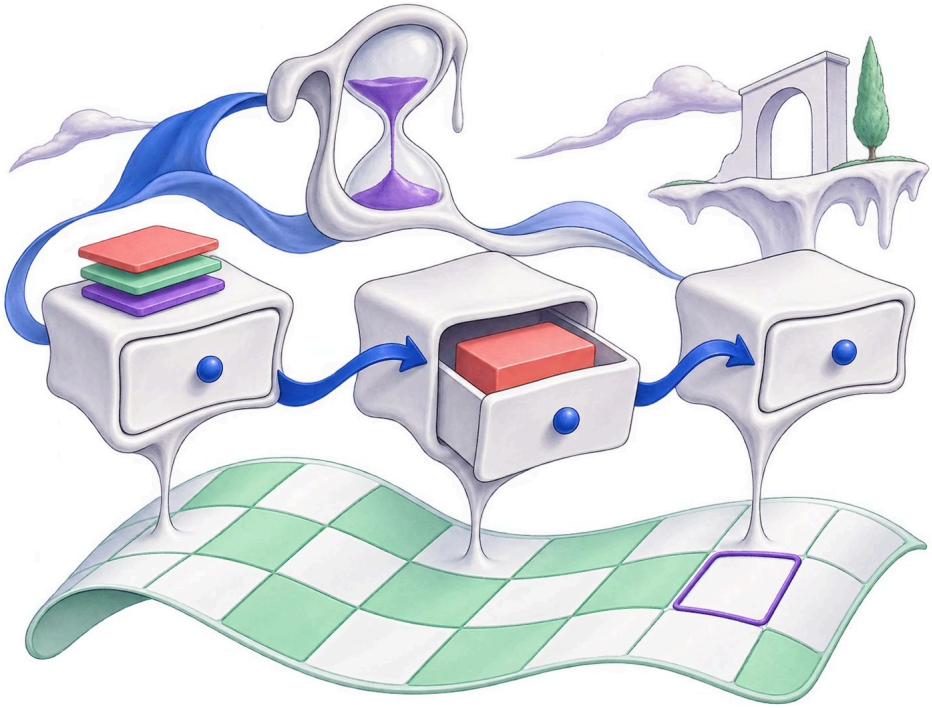


COURSE TEXTBOOK

Data Structures and Algorithms with C

Sercan Külcü

Giresun University · Department of Computer Engineering

Data Structures and Algorithms with C

Sercan Külcü
Giresun University, Department of Computer Engineering
Giresun, Türkiye

This book is the printed form of the course notes published at
`sercankulcu.github.io/teaching/data-structures-and-algorithms-with-c/`

Slides and study questions are on that page. The code for each chapter is in the `github.com/sercankulcu/data-structures-and-algorithms-c-en` repository.

© Sercan Külcü. All rights reserved.

Generated from the notes of 22 November 2026.

Preface

This book studies data structures and algorithms through their representation in memory and their implementation in C. Every topic starts from an abstract data type and arrives at a C99/C11 implementation that works with pointers, dynamic memory and structures, together with its complexity analysis. It is made up of the notes for the Data Structures and Algorithms with C course taught in the Department of Computer Engineering at Giresun University, and its chapters follow the order of the course.

The book has five parts. In order, they are **Foundations** (Chapters 1–3), **Linear Structures** (Chapters 4–9), **Trees** (Chapters 10–13), **Hashing, Graphs and Sets** (Chapters 14–18), and **Sorting** (Chapters 19–20).

Every chapter follows the same layout. It opens with learning outcomes. Formal definitions, theorems and worked examples are set in named boxes. *Common mistake* boxes point out the misconceptions students fall into most often. Traces show, step by step, how an operation changes the structure. Each chapter closes with a summary, a list of key terms, *Test Yourself* questions with their answers, suggestions for further reading and a bibliography. The glossary at the end of the book defines each key term in a sentence and names the chapter that explains it.

The code for each chapter is in its own folder in the companion repository, `github.com/sercankulcu/data-structures-and-algorithms-c-en`, and the programs use the same names as the listings in this book. Slides and study questions are on the course page.

Sercan Külcü
Giresun

Contents

Course Map 1

PART I · FOUNDATIONS

1 Introduction and Algorithm Analysis 5

1.1 Textbooks and love of the profession 6
1.2 The system life cycle 7
1.3 Algorithm 9
1.4 Abstract data type 11
1.5 Selection sort 17
1.6 Performance analysis 26
1.7 Asymptotic notation 36
1.8 The magic square 43
1.9 Measuring performance 47
1.10 Recursive routines 53
1.11 Complexity summary 55
1.12 Common mistakes 56
1.13 Summary 56
1.14 Key terms 57
1.15 Test Yourself 57
1.16 Further reading 59
1.17 References 60

2 Arrays and Structures 61

2.1 The array abstract data type 62
2.2 One-dimensional arrays in C 63
2.3 Dynamic arrays 67
2.4 Multidimensional arrays and address computation 74
2.5 Triangular matrices 80
2.6 Band matrices 84
2.7 Structures and unions 90
2.8 Strings 96
2.9 Matrix operations 99
2.10 Complexity summary 102
2.11 Common mistakes 103
2.12 Summary 104
2.13 Key terms 105
2.14 Test Yourself 105
2.15 Further reading 107
2.16 References 107

3 Recursion, Binary Search and Backtracking 108

3.1 Recursive algorithms 109
3.2 The call stack 111
3.3 Recursion or a loop? 115
3.4 Tail recursion 117
3.5 Binary search 119
3.6 The towers of Hanoi 126
3.7 Permutation generation 132
3.8 Brute force and backtracking 136
3.9 The satisfiability (SAT) problem 139
3.10 The n-queens problem 147
3.11 The knight's tour and Warnsdorff's rule 151
3.12 Complexity summary 154
3.13 Common mistakes 155
3.14 Summary 156
3.15 Key terms 156
3.16 Test Yourself 157
3.17 Further reading 159
3.18 References 159

PART II · LINEAR STRUCTURES

4 Linked Lists 161

4.1 When arrays are not enough 162
4.2 Pointers and dynamic memory in C 165
4.3 Singly linked list: nodes and traversal 169
4.4 Insertion 172
4.5 Deletion, search and releasing the list 180
4.6 Classic routines 187
4.7 The circular list and the header node 196
4.8 The doubly linked list 200
4.9 The available-node list 204
4.10 Complexity summary and measurement 208
4.11 Common mistakes 209
4.12 Exercises and solutions 210
4.13 Summary 213
4.14 Key concepts 214
4.15 Try It Yourself 214
4.16 Further reading 216
4.17 References 217

5 Polynomials and Sparse Matrices 218

5.1 The polynomial abstract data type 219
5.2 Storing a polynomial in an array 220
5.3 Polynomial addition: padd 224
5.4 Multiplication on arrays and operations on many polynomials 229
5.5 Polynomials as linked lists 232

5.6 Sparse matrices 240
5.7 Transposing 243
5.8 Sparse matrix multiplication 251
5.9 Other representations 256
5.10 Complexity summary 258
5.11 Common mistakes 259
5.12 Summary 259
5.13 Key terms 260
5.14 Test Yourself 260
5.15 Further reading 262
5.16 References 263

6 Stacks 264

6.1 The stack: last in, first out 265
6.2 The stack abstract data type 268
6.3 Array-based stack 271
6.4 Designing the stack as an object 278
6.5 Linked stack 285
6.6 Array-based or linked? 291
6.7 The call stack 295
6.8 Multiple stacks 297
6.9 Which output orders are possible? 301
6.10 Complexity summary 302
6.11 Common mistakes 303
6.12 Exercises and solutions 304
6.13 Summary 307
6.14 Key terms 308
6.15 Test Yourself 308
6.16 Further reading 310
6.17 References 311

7 Queues 312

7.1 What is a queue? 313
7.2 Linear queue in an array 316
7.3 Circular queue 321
7.4 Linked queue 332
7.5 Deques and applications 339
7.6 Complexity summary and measurements 348
7.7 Common mistakes 349
7.8 Exercises and solutions 350
7.9 Summary 353
7.10 Key terms 354
7.11 Test Yourself 354
7.12 Further reading 356
7.13 References 356

8 Stack and Queue Applications 357

8.1 The stack and backtracking 358
8.2 The maze problem 359
8.3 Expression notations 370
8.4 Evaluating a postfix expression 375
8.5 Converting infix to postfix 383
8.6 Prefix notation 389
8.7 Parenthesis checking 394
8.8 Other applications of stacks and queues 397
8.9 Complexity summary 402
8.10 Common mistakes 402
8.11 Exercises and solutions 403
8.12 Summary 407
8.13 Key terms 408
8.14 Test Yourself 408
8.15 Further reading 410
8.16 References 410

9 The List ADT and Generalized Lists 411

9.1 The list abstract data type 412
9.2 Implementation with an array 414
9.3 Linked implementation 421
9.4 Generalized lists 426
9.5 Basic recursive routines 434
9.6 Depth 445
9.7 Inversion 452
9.8 Building a tree with a list 454
9.9 Sharing, reference counting, and garbage collection 457
9.10 Exam questions 460
9.11 Complexity summary and measurements 463
9.12 Common mistakes 464
9.13 Exercises and solutions 464
9.14 Summary 467
9.15 Key terms 468
9.16 Test Yourself 468
9.17 Further reading 470
9.18 References 471

PART III · TREES

10 Trees 473

10.1 The tree data structure 474
10.2 Terminology 475
10.3 The binary tree 479
10.4 Representing a binary tree in memory 484
10.5 Recursive traversals 491

10.6 Traversal with a stack 500
10.7 Level-order traversal 509
10.8 Other tree procedures 514
10.9 Rebuilding a tree from traversals 520
10.10 Complexity summary 522
10.11 Common mistakes 523
10.12 Exercises and solutions 524
10.13 Summary 526
10.14 Key terms 526
10.15 Test Yourself 527
10.16 Further reading 529
10.17 References 529

11 Search Trees 530

11.1 The binary search tree 531
11.2 Insertion 535
11.3 Search 545
11.4 Depth 551
11.5 Deletion 557
11.6 Balanced trees 566
11.7 Exam questions 579
11.8 Complexity summary 584
11.9 Common mistakes 584
11.10 Exercises and solutions 585
11.11 Summary 587
11.12 Key terms 588
11.13 Test Yourself 588
11.14 Further reading 590
11.15 References 590

12 Threaded Binary Trees 592

12.1 Empty pointers and the idea of a thread 593
12.2 Building the threads 595
12.3 Inorder traversal 603
12.4 Inserting a node 612
12.5 Gains, costs, and a complexity summary 621
12.6 Common mistakes 621
12.7 Exercises and solutions 622
12.8 Summary 624
12.9 Key terms 625
12.10 Test Yourself 625
12.11 Further reading 627
12.12 References 627

13 Priority Queues and Heaps 628

13.1 Priority queues 629

13.2 Heap structure 631
13.3 Insertion 638
13.4 Deletion 646
13.5 Improving the implementation 652
13.6 Comparison with other structures 658
13.7 Bottom-up heap construction 661
13.8 d-ary heap 665
13.9 Applications and heap sort 667
13.10 Complexity summary 668
13.11 Common mistakes 668
13.12 Exam-style question 669
13.13 Exercises and solutions 670
13.14 Summary 671
13.15 Key terms 672
13.16 Test Yourself 672
13.17 Further reading 674
13.18 References 674

PART IV · HASHING, GRAPHS AND SETS

14 Hash Tables 677

14.1 The hashing idea 678
14.2 Hash functions 683
14.3 Open addressing 691
14.4 Chaining 701
14.5 Deletion, load factor and resizing 707
14.6 Chaining or open addressing? 716
14.7 Complexity summary 717
14.8 Common mistakes 718
14.9 Exercises and solutions 718
14.10 Summary 720
14.11 Key terms 720
14.12 Test Yourself 721
14.13 Further reading 723
14.14 References 723

15 Graphs 724

15.1 Introduction: bridges, maps and networks 725
15.2 Definitions and terminology 727
15.3 Implementing a graph in memory 732
15.4 Traversals 754
15.5 Components, cycles and bipartiteness 763
15.6 Complexity summary 772
15.7 Common mistakes 772
15.8 Exercises and solutions 773
15.9 Summary 775

15.10 Key terms 775
15.11 Test Yourself 776
15.12 Further reading 777
15.13 References 777

16 Sets and Union-Find 779

16.1 Disjoint sets 780
16.2 Array implementation 783
16.3 Analysis of the first version 788
16.4 The weighting rule 791
16.5 The collapsing rule 796
16.6 The Tarjan–van Leeuwen result and the Ackermann function 800
16.7 Applications 805
16.8 Exam question: the contents of the parent array 810
16.9 Complexity summary 811
16.10 Common mistakes 811
16.11 Exercises and solutions 812
16.12 Summary 813
16.13 Key terms 814
16.14 Test Yourself 814
16.15 Further reading 816
16.16 References 816

17 Minimum Spanning Trees 817

17.1 Spanning trees 818
17.2 The minimum-cost spanning tree 824
17.3 Kruskal's algorithm 828
17.4 Prim's algorithm 837
17.5 Other algorithms and comparisons 847
17.6 Exam-style question 850
17.7 Homework: coding Kruskal with data structures 852
17.8 Complexity summary 852
17.9 Common mistakes 853
17.10 Exercises 853
17.11 Summary 855
17.12 Key terms 856
17.13 Test Yourself 856
17.14 Further reading 858
17.15 References 858

18 Shortest Paths and Activity Networks 859

18.1 The shortest-path problem 860
18.2 The brute-force solution 861
18.3 Dijkstra's algorithm 863
18.4 Recovering the path: the predecessor array 872
18.5 Negative edges and all pairs 876

18.6 Activity networks and topological order 883
18.7 Activity-on-edge networks and the critical path 894
18.8 Earliest and latest times 897
18.9 Complexity summary 908
18.10 Common mistakes 909
18.11 Exercises and solutions 909
18.12 Summary 911
18.13 Key terms 912
18.14 Try It Yourself 912
18.15 Further reading 914
18.16 References 914

PART V · SORTING

19 Basic Sorting Algorithms 916
19.1 Sorting concepts 917
19.2 Selection sort 923
19.3 Insertion sort 926
19.4 Bubble sort 934
19.5 Inversions: a common measure 938
19.6 Counting sort 939
19.7 Shell sort 944
19.8 Bucket sort 947
19.9 A lower bound for comparison sorting 951
19.10 Comparing the algorithms 953
19.11 Common mistakes 956
19.12 Exercises and solutions 956
19.13 Summary 957
19.14 Key terms 958
19.15 Try it yourself 958
19.16 Further reading 960
19.17 References 961

20 Advanced Sorting Algorithms 962
20.1 Speeding up sorting: divide and conquer 963
20.2 Quicksort 964
20.3 Analysis of quicksort 970
20.4 Improving quicksort 973
20.5 The C library and hybrid sorts 981
20.6 Merge sort 984
20.7 Linked merge sort 991
20.8 Heap sort 995
20.9 Radix sort 1001
20.10 Comparison and choosing an algorithm 1007
20.11 Common mistakes 1008
20.12 Exercises and solutions 1009

20.13 Summary 1010
20.14 Key terms 1010
20.15 Try It Yourself 1011
20.16 Further reading 1012
20.17 References 1013

APPENDICES

A Glossary of Terms 1014

B Prerequisites for Data Structures and Algorithms with C 1023

C Important People in Data Structures and Algorithms with C 1039

Course Map

This page puts every chapter of the course on one map. Each chapter is a node, and an arrow shows which idea from an earlier topic the next one relies on. Read it before the course starts to see where the topics are heading, or during the course to see where you are.

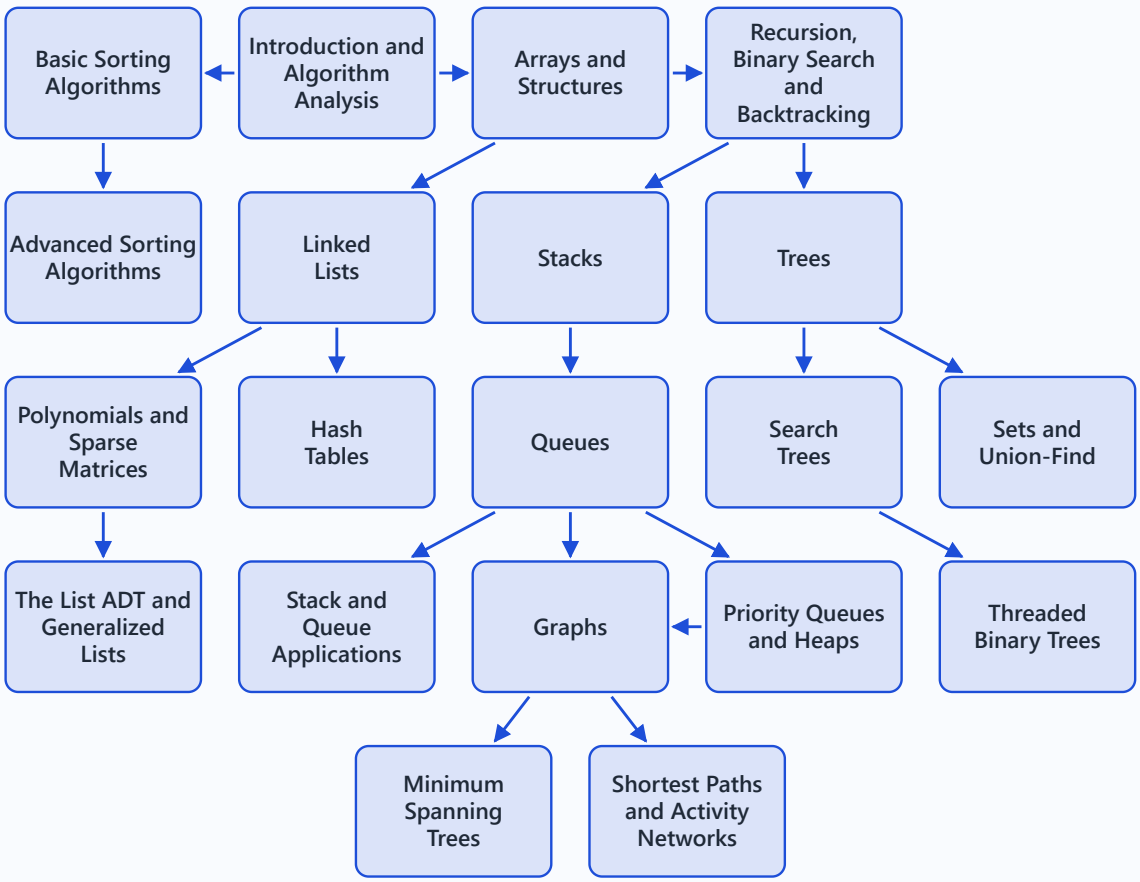


Figure 1: The course map. Each arrow points from a topic to one that builds on it. The paths that leave arrays and recursion split into linear structures, trees, graphs, and sorting.

Figure 1 shows the connections between the topics. Each node is a chapter, and an arrow indicates which idea of the earlier topic the later one relies on. Only the closest connections are drawn on the map; every chapter that a given chapter depends on is listed in Table 1.

Topic (chapter)	Builds on	Connection
Introduction and Algorithm Analysis (1)	—	The system life cycle, expressing an abstract data type with a C header file, and asymptotic analysis.
Arrays and Structures (2)	Introduction and Algorithm Analysis	The array ADT and the relation between arrays and pointers; dynamic arrays with malloc, and structures.
Recursion, Binary Search and Backtracking (3)	Introduction and Algorithm Analysis, Arrays and Structures	Tracing recursion with stack frames; binary search in a sorted array, Hanoi, and permutations.
Linked Lists (4)	Arrays and Structures	Nodes built with malloc and free where an array falls short.
Polynomials and Sparse Matrices (5)	Arrays and Structures, Linked Lists	Two representations of the same ADT: coefficient and term arrays, and linked lists; sparse matrices.
Stacks (6)	Arrays and Structures, Recursion, Binary Search and Backtracking, Linked Lists	The LIFO structure: push and pop with an array and a struct; the call stack is the stack of recursion.
Queues (7)	Stacks	The FIFO structure: the drifting problem of the linear queue is solved with the circular queue.
Stack and Queue Applications (8)	Stacks, Queues	Backtracking in a maze, converting an infix expression to postfix, and evaluating it.
The List ADT and Generalized Lists (9)	Linked Lists, Polynomials and Sparse Matrices	The contract of the list ADT; multivariate polynomials lead to the generalized list.
Trees (10)	Recursion, Binary Search and Backtracking, Linked Lists	Binary trees, array and linked representations; traversals are written recursively.
Search Trees (11)	Recursion, Binary Search and Backtracking, Trees	An ordering condition on the tree: insertion and search are the tree counterpart of binary search.
Threaded Binary Trees (12)	Trees, Search Trees	The n+1 null pointers become threads: inorder traversal without a stack.
Priority Queues and Heaps (13)	Queues, Trees	The priority form of the queue; the heap, by storing a complete binary tree in an array.
Hash Tables (14)	Arrays and Structures, Linked Lists	From direct addressing to hashing; open addressing uses an array, chaining uses linked lists.
Graphs (15)	Linked Lists, Queues, Priority Queues and Heaps	The adjacency list is a linked list; BFS traverses with a queue and DFS with a stack.
Sets and Union-Find (16)	Trees	Disjoint sets are stored as trees with a parent array; weighted union and path compression.
Minimum Spanning Trees (17)	Priority Queues and Heaps, Graphs, Sets and Union-Find	The greedy spanning tree: Kruskal uses union-find, Prim uses a heap.
Shortest Paths and Activity Networks (18)	Priority Queues and Heaps, Graphs	Dijkstra, Bellman-Ford, and Floyd-Warshall; topological order and the critical path in AOV and AOE networks.
Basic Sorting Algorithms (19)	Introduction and Algorithm Analysis, Arrays and Structures	The first example, selection sort, is generalized: insertion sort and bubble sort, stability.

Topic (chapter)	Builds on	Connection
Advanced Sorting Algorithms (20)	Recursion, Binary Search and Backtracking, Priority Queues and Heaps, Basic Sorting Algorithms	Quicksort and merge sort through divide and conquer; heap sort and radix sort.

Table 1: The connections on the map.

The backbone of the map can be read as **array and pointer → linked structure → tree → graph**. Every topic is carried from the abstract data type to a working C implementation, and its two tools are the array and the linked list built on malloc and free. Stacks and queues are written with both implementations and come together in the applications. Recursion leads to trees; heaps and disjoint sets are the array forms of trees. Graphs use all of these structures together: a heap and union-find for spanning trees, a priority queue for shortest paths. Sorting starts with selection sort in the first chapter and rests on divide and conquer and the heap in the last chapter.

PART I

Foundations

Chapters 1–3

PART I · CHAPTER 1

Introduction and Algorithm Analysis

A piece of software goes through a process that runs from writing down requirements to verifying the finished program. The two pillars of that process are data and the algorithm that works on it. These notes first describe the process and the notions of algorithm and abstract data type, and show how an abstract data type is built in C with a header file and an opaque type. They then use selection sort to introduce the first C code, the swap operation, and parameter passing through pointers. Most of the chapter is devoted to computing how much memory and time a program uses: counting program steps, classifying the result with O, Ω, and Θ notation, and comparing growth rates. The magic square problem shows that the right algorithm matters more than the hardware. The last sections cover measuring real running time with `clock()`, the pitfalls of measurement, and the cost of recursive routines.

LEARNING OUTCOMES

Know the process
Explain the five phases of the system life cycle and where data structure decisions fall in each.

Abstract
Define algorithm and abstract data type; build an interface in C with a header file and an opaque type.

Count
Derive the memory requirement and the number of program steps of a C routine (`sum` , `rsum` , `add`).

Classify
Express the step count with O, Ω, and Θ; tell the best, average, and worst cases apart.

Compare
Interpret how the common complexity classes separate as n grows.

Measure
Measure running time with `clock()` ; recognize measurement pitfalls such as clock ticks and output operations.

1.1 Textbooks and love of the profession

Data structures and algorithms are among the foundation courses on which the rest of computer engineering is built. Every engineer who writes an operating system, a compiler, a database, networking software, or a game engine decides how to keep data in memory and by what method to process it. In this course every structure is implemented in C. Because C exposes memory directly, it lets you see plainly how a structure sits in memory, what a pointer points to, and when allocated memory is given back.

1.1.1 Textbooks

The main textbook of the course is *Fundamentals of Data Structures in C* by Horowitz, Sahni, and Anderson-Freed. The order of topics, the examples, and most of the code in these notes come from the first chapter of that book. The book actually exists in three editions. The code in the first edition was written in a pseudo-language called SPARKS. In later editions the same material was rewritten in C and in C++. The topics and their order are nearly the same; what changes is the language of the code. The C++ edition is the most comprehensive because of its classes and templates, but this course follows the C edition (Figure 1.1).

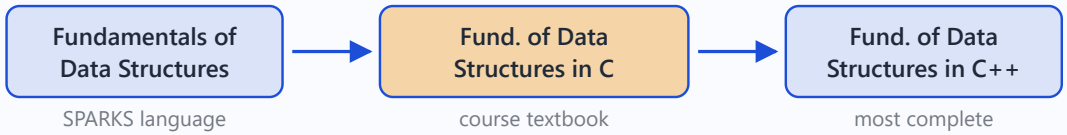


Figure 1.1: The three editions of the textbook. From left to right: the first edition written in the SPARKS pseudo-language, the C edition, which is the main book of the course, and the most comprehensive C++ edition.

Second on the shelf is Kernighan and Ritchie's *The C Programming Language*. Dennis Ritchie is the designer of C, and the book describes every detail of the language briefly and precisely. It is the place to look whenever there is doubt about the precedence of an operator, the return value of a standard library function, or the meaning of a declaration. It should therefore stay within reach as a desk reference.

Two more books are recommended as additional sources. *Introduction to Algorithms* by Cormen, Leiserson, Rivest, and Stein is known simply as CLRS, after its authors' initials. It is the most mathematically rigorous treatment of asymptotic notation, worst-case and average-case analysis, and amortized analysis. Sedgewick's *Algorithms in C* presents the same algorithms with C code and a large number of measurements.

1.1.2 Loving the profession

Doing a job well depends largely on loving it. An archaeologist lives in the present, but his mind is five thousand five hundred years in the past. Looking at a burial chamber, he can argue about why a pharaoh's wife might have been buried there. An astronomer takes photographs for years to show that a black hole exists, and rejoices when a star moves by one pixel in the image. A football fan travels eight hours, buys a ticket for a hundred and fifty liras, and braves the danger of stones and sticks. Most people find it odd when someone makes the same trip for a scientific conference. In all three examples, what makes people put up with the hardship is their interest in what they do.

Programming is no different. Someone who loves the profession explains things sincerely and gets listened to; nobody listens to someone who does the job carelessly. When writing code, too, you should be able to tell what is happening on every line, as if you were living inside the code. You should know what value a variable holds at that moment, which memory cell a pointer points to, and which iteration a loop is on. The trace tables given next to the code in these notes exist to build exactly this habit. A Chinese proverb says it briefly: the way to be happy for a lifetime is to love your work.

1.2 The system life cycle

A large piece of software is not written in one sitting. The road from an idea to a program that runs and can be trusted to be correct consists of successive phases. The whole set of these phases is called the **system life cycle**. Decisions about data structures and algorithms are made in particular phases of this cycle, so knowing the cycle shows where what you learn in this course fits in a real project.

1.2.1 Five phases

Horowitz and Sahni divide the life cycle into five phases (Figure 1.2):

1. **Requirements.** Every project begins with specifications that state its purpose. This phase writes down which inputs the programmer will be given and which outputs must be produced. For example, the input of a student records program is student records and grades, and its output is transcripts and grade point averages.
2. **Analysis.** The problem is broken into manageable parts. There are two ways to do this: bottom-up and top-down.
3. **Design.** The data objects of the problem and the operations on them are determined. The definition of the data objects corresponds to an *abstract data type*, and the definition of the operations corresponds to an *algorithm*. Both are independent of the programming language. The more time is spent on design, the fewer errors appear in later phases.
4. **Refinement and coding.** Representations for the data objects, that is, concrete data structures, are chosen, and the algorithms are coded on top of those structures. How the data is represented determines how efficient the algorithm will be. The same student list can be kept in an array or in a linked list; the cost of searching and inserting varies with this choice.

5. **Verification.** The correctness of the program is shown by proof and testing, and the errors found are removed. Each of these three activities is a research area in its own right.



Figure 1.2: The five phases of the system life cycle. If verification finds an error, one returns to the design or coding decisions, as the dashed arrow shows.

The dashed arrow in the figure shows that the cycle is not a one-way pipeline. An error found during verification sometimes calls for correcting a single line and sometimes for changing a data structure from scratch. The earlier the phase an error originates in, the more expensive it is to fix. A misunderstanding in the requirements causes far more work than a typo in the code.

1.2.2 Analysis: bottom-up and top-down

There are two ways to break up the problem in the analysis phase.

In the **bottom-up** approach the problem is divided directly into parts, and each part is focused on separately. The modules are seen from the start; the problem is solved module by module, and then the parts are combined. This route is fast for teams that have done a similar project before and know which modules are needed. It is preferred when there is experience with the subject of the project.

In the **top-down** approach a master plan is made first. One asks what the goal is and what the final product will look like. According to this plan, the problem is divided into subproblems, and the subproblems into smaller pieces. This route takes more time, but the likelihood of backtracking and error is lower. It is preferred when there is no experience with the subject, because the parts are defined knowing their place in the whole.

Example 1.1

Let a student records program be analyzed top-down. At the highest level the program consists of three jobs: record management, grade entry, and reporting. Reporting splits in turn into producing transcripts and computing averages. Computing an average requires finding all the courses of a student and taking a credit-weighted mean. At every level, the parts are defined so that they serve the job one level above. In a bottom-up analysis, by contrast, one directly writes modules such as "add student record", "enter grade", and "compute average", and then joins them with a menu.

1.2.3 Postponing design and implementation decisions

The decisions made in the design phase are independent of the language. For example, the fields of the student record (number, name, department, grade list) are decided in this phase. But whether this record will be implemented as an array, a linked list, or a tree can be postponed. Postponing is not laziness but a deliberate choice. As it becomes clearer which operations the program will perform and how often, the most suitable implementation can be chosen better. If records are mostly looked up by number, a sorted array or a search tree is suitable. If insertions and deletions are frequent, a linked list comes to the fore.

The same thought applies to coding itself: the longer coding is delayed, the better a method can be found in the background. Sitting down at the keyboard without thinking about the problem means setting out with the first method that comes to mind, which is often not the best one.

1.2.4 Coding and verification

Verification covers four activities.

- **Correctness proofs.** The correctness of a program can be proved with mathematical techniques. For a small routine this is possible, but proving a large project from start to finish is almost impossible. The practical way is to use algorithms whose correctness has been proved before; this reduces the chance of errors.
- **Testing.** Testing needs working code and test data that covers all the scenarios. Empty input, single-element input, the largest input, sorted and reverse-sorted input, and repeated values are at the top of the list. That the compiler reports no syntax errors and the program gives correct output on one example does not show that the program is correct.
- **Debugging.** If a proof or tests reveal an error, it is tracked down by considering the design and coding decisions. A well-documented design makes this job considerably easier.
- **Running time.** A program that is correct but runs slowly is also of little value. Correctness and efficiency are separate questions; each must be answered separately.

> **Common mistake**
>
> Thinking "it compiled and ran, so it is correct". A sorting program may give the right result on ten random numbers, yet crash on an empty array or sort incorrectly on repeated values. The test data must cover not only the "ordinary" input but also the edge cases.

1.3 Algorithm

In the design phase the operations are defined as an algorithm. This word, which in everyday language means "a way of doing a job", has a precise definition in computer science.

1.3.1 Definition and the five criteria

> **Definition 1.1**
>
> An **algorithm** is a finite sequence of instructions that accomplishes a specific task. Every algorithm satisfies five criteria: input, output, definiteness, finiteness, and effectiveness.

The five criteria are summarized in Table 1.1.

Criterion	Meaning
Input	Zero or more quantities are supplied from outside.
Output	At least one quantity is produced.
Definiteness	Each instruction is clear and unambiguous.
Finiteness	In every case, it terminates after a finite number of steps.
Effectiveness	Each instruction is basic and feasible enough to be carried out with pencil and paper.

Table 1.1: The five criteria an algorithm must satisfy.

Let us take the criteria one at a time. The **input** criterion allows zero inputs: an algorithm that says "write the first ten primes" takes nothing from outside. The **output** criterion, on the other hand, demands at least one result; a method that produces nothing has no useful task.

Definiteness requires each instruction to be understood in only one way. The instruction "add 6 or 7 to x" is not definite, because it does not say which one is added. "Multiply the number by a sufficiently large value" is not definite either. A programming language such as C satisfies this criterion automatically: every construct of the language has a defined meaning. In an algorithm written in natural language, however, definiteness needs separate attention.

Finiteness requires the algorithm to stop at some point for every input. The instruction "start at 1, keep doubling the number, stop when the result is negative" never stops in mathematics. **Effectiveness** requires every instruction to be actually doable. Adding two integers is an effective operation. The instruction "write down all the digits of π" is definite, but it is not effective because it cannot be done in finite time.

1.3.2 Algorithm versus computational procedure

The finiteness criterion has an interesting consequence: programs that are not supposed to terminate are not algorithms. An operating system runs as long as the computer stays on; it waits for a task, executes it, and goes back to waiting. Because it never ends, by definition it does not count as an algorithm. Such programs are called **computational procedures**. The parts inside the operating system, however, such as a routine that reads a file from disk or one that picks the next job among those waiting, are algorithms because they terminate.

An algorithm can be expressed in natural language, as a flowchart, or in a programming language. Natural language is easy to read but has trouble achieving definiteness. A flowchart is illustrative for small algorithms, while for large ones it turns into a tangled drawing. In this course algorithms are written in C. In places, short C-like pseudocode is used to show an idea without going into the details of the code.

1.4 Abstract data type

In the design phase, data objects are defined independently of the language. The tool for this definition is the abstract data type. An abstract data type separates what a piece of data "does" from "how it is done". This separation is the common framework of every structure implemented in this course (stack, queue, list, tree, graph).

1.4.1 Data type and abstract data type

Consider the type `int` in C. This type consists of two things: the integers in a certain range, and the operations defined on those integers (`+`, `-`, `*`, `/`, `%`, comparisons, assignment). A programmer who uses an `int` variable does not need to know how many bits the number occupies in memory or with what encoding. Nor does the programmer know how addition is carried out in the processor. Only what the operations do is known. In general, a **data type** is a collection of objects together with the operations defined on those objects.

> **Definition 1.2**
>
> An **abstract data type** (ADT) is a data type in which the **specification** of the objects and operations is separated from the representation of the objects in memory and from the **implementation** of the operations.

Two notions face each other in this definition.

- **The specification answers the question "WHAT is done?"** It gives the names of the operations, the types of their parameters, the type of their results, and what each operation does. A programmer who uses the ADT knows only this.
- **The implementation answers the question "HOW is it done?"** It contains the representation of the data in memory (a `struct`, an array, a linked list) and the code that performs the operations. It is hidden from the user.

The benefit of the separation is concrete. Because the implementation is hidden, it can be changed freely as long as the specification stays the same. A stack written with an array can later be rewritten with a linked list, and not a single line of the program that uses the stack changes. Nor can the user

write code that depends on the internals of the implementation, because those internals are not visible. The same ADT can be implemented with several data structures, and the choice can be left to the refinement phase of the life cycle.

The operations of an ADT are grouped by their roles:

- **Constructor** operations create a new instance of the type.
- **Transformer** operations produce a new value from existing instances.
- **Observer** or **reporter** operations give information about an instance without changing it.

1.4.2 Specification of the Fraction ADT

Let us make the notion of an abstract data type concrete with a small but complete example: fractions. A fraction consists of a numerator and a nonzero denominator. Let operations be defined on fractions for creating, adding, testing equality, printing, and deleting. The specification is written in a language-independent notation as follows:

```
ADT Fraction
  objects: num/den, den ≠ 0
  operations:
    create(n, d)   → Fraction /* constructor */
    add(a, b)      → Fraction /* transformer */
    equal(a, b)    → Boolean  /* observer */
    print(a)                  /* reporter */
    delete(a)
```

The first line of the specification defines the objects: values consisting of a numerator and a denominator, with the denominator nonzero. The following lines give the operations and the result type of each. `create(n, d)` is a constructor. `add(a, b)` is a transformer because it produces a new fraction from two fractions. `equal(a, b)` is an observer that returns a truth value; `print(a)` is a reporter that writes the fraction to the screen. `delete(a)` gives back the resource the fraction occupies. There is not a single word in the specification about how the fraction will be kept in memory, and there does not need to be.

The specification also contains a **precondition**: the denominator cannot be zero. This condition makes it necessary to decide what the `create` operation will do when it receives a request with a zero denominator. That question is answered in the implementation.

1.4.3 The interface in C: header file and opaque type

C has no notion of a class, but it offers two tools for separating the specification from the implementation: the **header file** and the **incomplete type**. A header file (`.h`) carries type and function declarations and is included into other files with `#include` . The implementation sits in a separate source file (`.c`). The interface of the Fraction ADT is the following:

```c
/* fraction.h : interface (WHAT?) */
#ifndef FRACTION_H
#define FRACTION_H
typedef struct fraction Fraction; /* opaque */

Fraction *fraction_create(int num, int den);
Fraction *fraction_add(const Fraction *a,
                       const Fraction *b);
int  fraction_equal(const Fraction *a,
                    const Fraction *b);
void fraction_print(const Fraction *f);
void fraction_delete(Fraction *f);
#endif
```

The lines of the file work as follows:

- `#ifndef FRACTION_H`, `#define FRACTION_H`, and the final `#endif` form an **include guard**. These three lines address the preprocessor. If the file is brought in a second time with `#include` in the same translation unit, `FRACTION_H` is already defined, so the content is skipped and the same declarations are not made twice.
- The line `typedef struct fraction Fraction;` does two things. First it declares that a structure named `struct fraction` exists, but does not say what its fields are. Then it gives this structure the short name `Fraction`. A structure whose fields are unknown is an **incomplete type**: the compiler does not know its size. Therefore a file that uses `fraction.h` cannot define a variable of type `Fraction` or access a field such as `f->num`. It can only hold a pointer of type `Fraction *`. A type whose fields are hidden from the user in this way is called an **opaque type**.
- The five function declarations correspond to the five operations of the ADT. Fractions are always passed around by pointer. The operations that only read a fraction take parameters of type `const Fraction *`; this is a promise, checked by the compiler, that the function will not modify the fraction pointed to.

The first part of the implementation contains the fields of the structure and the constructor:

```c
/* fraction.c : implementation (HOW?) */
#include <stdio.h>
#include <stdlib.h>
#include "fraction.h"

struct fraction { int num, den; };

Fraction *fraction_create(int num, int den) {
  Fraction *f;
  if (den == 0) return NULL;
  f = malloc(sizeof *f);
  if (f == NULL) return NULL;
  f->num = num;
  f->den = den;
  return f;
}
```

The line `struct fraction { int num, den; };` completes the structure. Because this line appears only inside `fraction.c`, the fields are known only in this file. The opaque type approach is C's counterpart of the ADT principle that "the implementation is hidden".

The function `fraction_create` does the following, line by line:

1. If the denominator is zero, the fraction cannot be created; the function returns `NULL`. `NULL` is the **null pointer**, which points to no object, and here it means "failure".
2. The call `malloc(sizeof *f)` allocates memory of the size of one `struct fraction`. `malloc` allocates a block of memory of the requested number of bytes from the **heap** region and returns the starting address of the block. The expression `sizeof *f` gives the size of the type that `f` points to, that is, of `struct fraction`. Using `*f` instead of rewriting the type name keeps the size automatically correct if the type of `f` changes later. `sizeof` does not evaluate its operand; the expression is safe even though `f` has not yet received a value.
3. If no memory is left, `malloc` returns `NULL`. This case is checked, and the function returns `NULL` as well. Without the check, the next line would try to write to the place a null pointer points to, and the program would crash.
4. The `->` in the line `f->num = num;` is the **arrow operator**: it accesses a field of the structure the pointer points to. `f->num` means the same as `(*f).num`.
5. Finally the address of the new fraction is returned.

The local variable `f` of the function disappears when the function returns. But the block allocated with `malloc` lives on, because a block on the heap stays allocated until it is given back with `free`. Since the address the function returns points to this block, the caller can use the fraction. The situation would be different if the address of a local variable were returned: once the function's frame is gone, that address becomes invalid.

Common mistake

Not checking the return value of `malloc`. In small test programs memory almost never runs out and the error stays invisible. But in a program that works with large inputs or stays up for a long time, `malloc` can return `NULL`. Unchecked code then crashes with a write such as `NULL->num`. After every `malloc` call, the result must be compared with `NULL`.

1.4.4 The rest of the implementation

The rest of `fraction.c` contains the remaining four operations:

```
Fraction *fraction_add(const Fraction *a,
                       const Fraction *b) {
  return fraction_create(
      a->num * b->den
        + b->num * a->den,
      a->den * b->den);
}
int fraction_equal(const Fraction *a,
                   const Fraction *b) {
  return a->num * b->den
      == b->num * a->den;
}
void fraction_print(const Fraction *f) {
  printf("%d/%d\n", f->num, f->den);
}
void fraction_delete(Fraction *f) { free(f); }
```

`fraction_add` applies the common-denominator formula: $a/b + c/d = (a·d + c·b) / (b·d)$. It does not allocate the result itself but calls `fraction_create`. That way the denominator check and the memory allocation stay in a single place. Since both denominators are nonzero, their product is nonzero too; `fraction_create` returns `NULL` only if memory runs out. The implementation does not reduce the result: the result of `1/2 + 1/2` is `4/4`. If reduction is wanted, a line can be added that divides the result by the greatest common divisor; this addition does not change the interface.

`fraction_equal` tests the equality of two fractions by cross-multiplication: $a/b = c/d$ if and only if $a·d = c·b$. This method also works on unreduced fractions; `2/4` and `1/2` are found equal. In C a comparison operator yields 1 or 0 of type `int`; the function therefore returns the result of the comparison directly. The `Boolean` of the specification is represented in C by this `int` value.

`fraction_print` writes the fraction in the form `num/den`. `fraction_delete` gives the fraction's block back to the heap with `free`. `free` returns a block allocated with `malloc` to the system. Standard C guarantees that the call `free(NULL)` does nothing; `fraction_delete` can therefore be called safely with a null pointer as well.

Common mistake

Working with very large numerators and denominators. Products such as `a->num * b->den` can exceed the range of `int`. Signed integer overflow is *undefined behavior* in C: the result can be an arbitrary value, and the program can behave unexpectedly. The problem does not show in small examples; in an implementation that works with large values one must use `long long` or reduce the result at every step.

1.4.5 Using the interface: contract and ownership

A program that uses fractions knows only the interface:

```c
/* main.c : knows only the interface */
#include "fraction.h"
#include <stddef.h> /* NULL */
int main(void) {
  Fraction *a = fraction_create(1, 2);
  Fraction *b = fraction_create(1, 3);
  Fraction *t = NULL;
  if (a && b) t = fraction_add(a, b);
  if (t) fraction_print(t);  /* 5/6 */
  fraction_delete(a); fraction_delete(b);
  fraction_delete(t);  /* free(NULL) is harmless */
  return 0;
}
```

`main.c` includes only `fraction.h` and the header `<stddef.h>` for the constant `NULL`. It does not see the fields of `struct fraction`. The program consists of three files, and when compiling, the two source files are given together, for example `cc fraction.c main.c`. The compiler compiles each `.c` file separately, and the linker binds the calls in `main.c` to the definitions in `fraction.c`.

The flow of the program is traced in Table 1.2. The addresses in the table are example values chosen to show where the blocks sit; real addresses may differ on every run.

Line	Event	`a`	`b`	`t`
`a = fraction_create(1, 2)`	a block {1, 2} is allocated on the heap	0x2000 → 1/2	—	—
`b = fraction_create(1, 3)`	a block {1, 3} is allocated on the heap	0x2000	0x2010 → 1/3	—
`t = NULL`	no result yet	0x2000	0x2010	NULL
`if (a && b)`	neither is NULL: true	0x2000	0x2010	NULL
`t = fraction_add(a, b)`	numerator 1·3 + 1·2 = 5, denominator 2·3 = 6; a new block	0x2000	0x2010	0x2020 → 5/6
`if (t) fraction_print(t)`	`5/6` is printed	0x2000	0x2010	0x2020

Line	Event	`a`	`b`	`t`

`fraction_delete(a);`
`fraction_delete(b);`
`fraction_delete(t)`
	three blocks return to the heap	invalid	invalid	invalid

Table 1.2: Trace of the program `main.c` (addresses are examples).

After the last lines of the program, the variables `a`, `b`, and `t` still carry their old addresses, but those addresses no longer point to a valid object. Such pointers are called **dangling pointers**. Using them is undefined behavior. Since the program ends immediately, no problem arises here.

Two rules must be written down explicitly in order to use an interface safely.

- **Contract.** `fraction_create` returns `NULL` if the denominator is zero or memory has run out. The caller is obliged to check this result. The checks `if (a && b)` and `if (t)` in `main` follow from this contract. Without the checks, `fraction_add(a, b)` might try to read a field of a null pointer.
- **Ownership.** Every fraction returned by `fraction_create` and `fraction_add` belongs to the caller. When the caller is done with the fraction, it must release it with `fraction_delete`. A block that is not released stays on the heap until the program ends. Memory that can no longer be reached through any pointer but has not been given back is called a **memory leak**. A program that creates a fraction on every iteration of a loop and forgets to delete it grows until memory runs out.

The reason `t` is initialized to `NULL` is also the ownership rule. If `a` or `b` cannot be created, `fraction_add` is never called and `t` remains `NULL`. In that case the call `fraction_delete(t)` at the end of the program turns into `free(NULL)` and is harmless. Without an initial value, `t` could hold an arbitrary address, and the call to `free` could crash the program.

Common mistake

Trying to write `a->num` in `main.c`, or to define a variable such as `Fraction f;`. Both give compile errors, because for `main.c` the `struct fraction` is an incomplete type. This error is not a flaw but the very purpose of the opaque type: the user is forced to work only through the functions in the interface. Another mistake is deleting the same fraction twice. The second `free` tries to give back a block that is no longer allocated; this is also undefined behavior.

1.5 Selection sort

Let us see the notion of an algorithm, and how it is written in C, for the first time on a concrete problem: sorting an array of integers from smallest to largest. **Sorting** means putting elements into ascending or descending order by their keys. The problem has many solutions. In this section selection sort, one of the easiest to understand, is presented first as an idea, then as pseudocode, and finally as C code. The step-by-step conversion of an idea into code is the usual course of algorithm design.

1.5.1 The idea

Let an unsorted array `list` of n elements be given. **Selection sort** rests on this idea: find the smallest element in the unsorted part of the array and put it at the front of that part. Then the unsorted part shrinks by one element and the same job is repeated. In the first round the smallest of the whole array settles into `list[0]`, in the second the smallest of the rest into `list[1]`. In general, in round i the smallest of the range `list[i]` ... `list[n-1]` is moved to `list[i]`. In C-like pseudocode the idea is written as follows:

```
for (i = 0; i < n; i++) {
    examine list[i] .. list[n-1];
    let the smallest integer be list[min];
    swap list[i] and list[min];
}
```

The second and third lines of this pseudocode are not yet C; expressions such as "examine the range" and "let the smallest element be" are not commands a C compiler would understand. But the idea is clear, and each of these lines is easily translated into C. To find the smallest element, the range is scanned from left to right; the index of the smallest seen so far is kept in a variable `min`. The swap is done with three assignments.

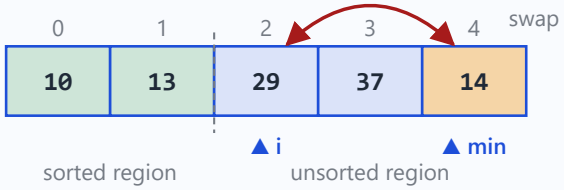


Figure 1.3: An intermediate state of selection sort. The green cells (10, 13) are the sorted region and are in their final places. For `i = 2` the smallest element of the unsorted region (29, 37, 14) is 14, at index 4; `list[2]` and `list[4]` are swapped.

Figure 1.3 shows the state the algorithm maintains in every round. The array is divided into two regions by a dashed line. The **sorted region** on the left holds the i smallest elements of the array in order; these elements never move again. The **unsorted region** on the right holds the remaining elements. Each round grows the sorted region by one element. This holds at the start of every round and is the justification for the algorithm's correctness. After $n - 1$ rounds the sorted region contains the $n - 1$ smallest elements; the one element left is the largest, so it already sits in the last cell. The array is sorted.

A written algorithm does not have to be accepted as it is. Understanding it, seeing its weak points, and being able to propose a better one when needed is the level this course aims at. Selection sort will be examined with this eye: how many comparisons does it make, how many swaps, and does a sorted input make its job easier?

1.5.2 C implementation

First the main program that uses the sorting function:

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 101
#define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
void sort(int list[], int n);

int main(void) {
  int i, n, list[MAX_SIZE];
  printf("Enter the array size: ");
  if (scanf("%d", &n) != 1 || n < 1 || n > MAX_SIZE) {
    printf("Invalid n!\n");
    return 1;
  }
  for (i = 0; i < n; i++) list[i] = rand() % 100;
  sort(list, n);
  for (i = 0; i < n; i++) printf("%d ", list[i]);
  return 0;
}
```

The main program works as follows, line by line:

- `#define MAX_SIZE 101` gives the maximum size of the array as a constant. When the constant is changed, both the size of the array and the bounds check change automatically.
- `#define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))` is a **macro**. A macro is a name defined with `#define` that is replaced as text before compilation. Where `SWAP(list[i], list[min], temp)` is written, the preprocessor puts the text `((temp) = (list[i]), (list[i]) = (list[min]), (list[min]) = (temp))`. The comma operator performs the three assignments in order from left to right.
- `void sort(int list[], int n);` is a **function declaration** (a prototype). Because the definition of `sort` comes after `main`, the parameter types and return type of the function are declared to the compiler in advance.
- `scanf("%d", &n)` reads the array size from the user. `&n` is the address of `n`; `scanf` writes the value it reads to this address. `scanf` returns the number of items it read successfully. If the user types a letter instead of a number, a value other than 1 is returned.
- The condition rejects three cases: the read failed, `n < 1`, or `n > MAX_SIZE`. The last check matters. The array `list` has `MAX_SIZE` cells; if a larger `n` were accepted, the loops would write to the memory after the end of the array. C does not check for such overflows.
- `rand() % 100` produces a pseudorandom number between 0 and 99. Because `srand` is not called, the program produces the same array on every run; this is a convenience so that results can be reproduced.

- In the call `sort(list, n)`, not the array itself but the address of its first element is passed. The expression `list[i]` inside `sort` accesses the memory starting at this address; therefore the changes `sort` makes to the array change the array in `main`.

The sorting function is the direct C counterpart of the pseudocode:

```c
void sort(int list[], int n) {
  int i, j, min, temp;
  for (i = 0; i < n - 1; i++) {
    min = i;
    for (j = i + 1; j < n; j++)
      if (list[j] < list[min])
        min = j;
    SWAP(list[i], list[min], temp);
  }
}
```

- The outer loop runs from `i = 0` to `n - 2`. In the pseudocode the loop ran `n` times; the last round is unnecessary, because a one-element range is already sorted. That is why the loop condition is written `i < n - 1`.
- `min = i;` provisionally takes the first element of the range to be the smallest.
- The inner loop scans the remaining elements of the range from `j = i + 1` to `n - 1`. If `list[j] < list[min]`, a smaller element has been found and `min` is updated. When the inner loop finishes, `min` is the index of the smallest element of the range.
- `SWAP(list[i], list[min], temp)` moves this element to the front of the range. If `min` already equals `i`, the swap exchanges an element with itself; the result does not change, and only three unnecessary assignments are made.

When the program is compiled and run with the input `n = 10`, it prints the ten randomly generated numbers in sorted order. In the environment in which these notes were prepared, the output was `0 24 34 41 58 62 64 67 69 78`. The sequence that `rand()` produces depends on the C library; on another compiler the numbers come out different, but they are always printed in ascending order.

1.5.3 Code trace: {29, 10, 14, 37, 13}

Let us trace the function `sort` on the five-element array {29, 10, 14, 37, 13}. Table 1.3 gives, for every round, the `min` value found by the inner loop and the array after the swap.

`i`	Comparisons of the inner loop	`min`	Swap	Array
0	10 < 29 yes (min = 1); 14 < 10 no; 37 < 10 no; 13 < 10 no	1	list[0] ↔ list[1]	{**10**, 29, 14, 37, 13}
1	14 < 29 yes (min = 2); 37 < 14 no; 13 < 14 yes (min = 4)	4	list[1] ↔ list[4]	{**10**, **13**, 14, 37, 29}
2	37 < 14 no; 29 < 14 no	2	with itself (no change)	{**10**, **13**, **14**, 37, 29}

`i`	**Comparisons of the inner loop**	`min`	**Swap**	**Array**
3	29 < 37 yes (min = 4)	4	list[3] ↔ list[4]	{**10**, **13**, **14**, **29**, 37}

Table 1.3: Trace of the function `sort` on the array {29, 10, 14, 37, 13}.

In the first round the smallest element, 10, is found at index 1 and exchanged with 29. In the second round the scan first takes 14 and then 13 to be the smallest; 13 is swapped with the 29 at index 1. In the third round 14 is already the smallest of the range; `SWAP` runs but changes nothing. In the fourth and last round 29 and 37 are exchanged. When the outer loop reaches `i = 4`, the condition `i < n - 1` is false and the function ends. In total *4 + 3 + 2 + 1 = 10* comparisons were made and `SWAP` was called four times, in three of which two different cells were actually exchanged.

1.5.4 Analysis

The number of comparisons selection sort makes does not depend on the values of the input; it depends only on n. In round i the inner loop takes the values $j = i + 1, \ldots, n - 1$, that is, it makes $n - 1 - i$ comparisons. The sum over all rounds is:

> $(n - 1) + (n - 2) + \ldots + 1 = n(n - 1) / 2$

This sum is the sum of the first $n - 1$ natural numbers. The formula is verified as follows: write the sum once forwards and once backwards and add term by term; every pair adds up to n and there are $n - 1$ pairs, so twice the sum is $n(n - 1)$. For $n = 5$ one gets $5 \cdot 4 / 2 = 10$ comparisons, which is the number in the trace table. For $n = 1000$ about half a million comparisons are needed. Because the number of comparisons grows in proportion to n^2, the time of selection sort is $O(n^2)$. The notation $O(n^2)$ says that for large n the number of steps does not exceed a constant multiple of n^2.

The number of swaps, on the other hand, is small: a single `SWAP` is performed in each round, $n - 1$ times in total. Hence at most $n - 1$ swaps take place. If the elements are large records and moving them is expensive, this property makes selection sort attractive. As extra memory the algorithm uses only the variables `i`, `j`, `min`, and `temp`. The array is sorted in place.

1.5.5 A faulty version of sort()

When people try to write selection sort, a commonly seen code is this:

```
void sort(int list[], int n) {
  int i, j, min, temp;
  for (i = 0; i < n; i++) {
    min = i;
    for (j = 0; j < n; j++)
      if (list[j] > list[min]) {
        min = j;
        SWAP(list[i], list[min], temp);
      }
  }
}
```

Surprisingly, this code still sorts the array. While preparing these notes, all permutations for $n \le 7$ were tried; the result came out sorted in every one. But the code is not the selection sort of the pseudocode. There are three differences:

1. The inner loop starts not at `j = i + 1` but at `j = 0`; it scans the whole array in every round.
2. The condition is `>`, not `<`; what is sought is not the smallest but the largest element so far.
3. The swap is made not once after the inner loop but inside the inner loop, every time the condition is true.

Even though the result is correct, the cost is very different. Table 1.4 gives the numbers of comparisons and swaps that the two versions make on the same inputs. The values were measured by adding counters to the two versions. The swap column counts only the `SWAP`s in which two different cells were actually exchanged (`i ≠ min`). The random row was taken from a sample array of one hundred elements generated with `rand() % 100`; on another random array the swap counts change slightly, while the comparison counts do not.

Input	Version	Comparisons	Swaps
{29, 10, 14, 37, 13}	faulty	25	7
	selection	10	3
100 random	faulty	10000	2561
	selection	4950	94
500, 499, ..., 1 (descending)	faulty	250000	124750
	selection	124750	250

Table 1.4: Numbers of comparisons and swaps of the faulty version and of selection sort.

The faulty version makes n comparisons in every round and runs n rounds: n^2 in total. This is more than twice the $n(n - 1)/2$ comparisons of selection sort. The real difference is in the swaps. On the descending array of 500 elements the faulty version makes 124,750 swaps; selection sort makes only 250. On a descending array, every swap of selection sort puts two elements into their final places at

once. For example, the first swap exchanges 1 and 500. That is why 250 swaps are enough. On this input the number of swaps of the faulty version also equals the number of pairs of elements in reverse order, that is, *500 · 499 / 2 = 124,750*.

This example shows an important lesson: a program giving the correct result does not show that it implements the intended algorithm. Algorithm analysis asks, besides the result, what it costs to reach that result. The correct version, that is, the one that starts at `j = i + 1` and swaps after the inner loop, is the version used in this chapter.

1.5.6 SWAP: macro or function?

Exchanging the values of two variables is called a **swap**. A swap can be written in two ways in C. The first is the macro used in the program:

```
#define SWAP(x, y, t) \
        ((t) = (x), (x) = (y), (y) = (t))

int temp;
SWAP(list[i], list[min], temp);
```

The backslash (`\`) at the end of the line tells the preprocessor that the macro definition continues on the next line. The advantage of the macro is that it carries no function call cost. The preprocessor puts the macro text in place of the call; at run time there is no call, parameter copying, or return. On the other hand, the temporary variable `t` must be supplied to the macro from outside; this is why `temp` is defined separately inside `sort` .

The second way is a function:

```
void swap(int *x, int *y) {
  int temp = *x;
  *x = *y;
  *y = temp;
}

swap(&list[i], &list[min]);
```

The parameters of the function are not integers but pointers to integers (`int *`). In C, parameters are passed **by value**: the function receives a copy of the value sent. A function written as `swap(int x, int y)` swaps only its own copies; the variables on the caller's side do not change. That is why not the variables themselves but their addresses are sent: `swap(&list[i], &list[min])` . Inside the function, the expression `*x` gives the value at the place `x` points to. Here `*` is the **dereference operator**. The assignment `*x = *y` writes the value that `y` points to into the place that `x` points to.

The advantage of the function is that the compiler does type checking: if `swap` is given an address that is not an `int`, the compiler warns. The temporary variable `temp` is a local variable of the function; the caller does not need to define anything. The function's name is given in lowercase as `swap`. Had the function also been named `SWAP`, the preprocessor would try to expand every occurrence of `SWAP(` as the macro and the two definitions would collide.

The macro has a hidden danger: it evaluates its arguments more than once. When `SWAP(a[i++], a[j], t)` is written, the preprocessor produces this text:

```
((t) = (a[i++]), (a[i++]) = (a[j]), (a[j]) = (t))
```

`i++` is executed twice. If `a = {10, 20, 30, 40}`, `i = 0`, and `j = 3`, first `t = a[0] = 10` and `i` rises to 1. Then `a[1] = a[3] = 40` is written and `i` rises to 2. Finally `a[3] = 10`. The result is `{10, 40, 30, 10}` and `i = 2`: no swap has been made, and the value 20 has been lost. Had the same call been made with the function, as `swap(&a[i++], &a[j])`, `i++` would be executed once and `a[0]` and `a[3]` would be swapped correctly. These results were verified by compiling and running the example.

Common mistake

Giving the macro an argument with a side effect: `SWAP(a[i++], a[j], t)`, `SWAP(x, f(), t)`, and so on. A macro is not a function; arguments are copied as text and re-evaluated everywhere they appear. Macro arguments must be plain variables or expressions without side effects. This is also why every parameter in the macro definition is wrapped in parentheses: expressions such as `SWAP(a[k], b[k + 1], t)` are then grouped correctly.

1.5.7 swap() in memory, step by step

Let us trace how parameter passing with pointers works in memory, using a small program:

```c
void swap(int *x, int *y) {
  int temp = *x;
  *x = *y;
  *y = temp;
}

int main(void) {
  int a = 3, b = 8;
  swap(&a, &b);
  printf("%d %d\n", a, b);
}
```

When run, the program prints `8 3`. To see how this result is reached, let us follow the memory step by step. Every function call opens a **stack frame** on the **call stack**. The frame holds the parameters and local variables of that call and disappears when the call ends.

1. **Start.** In the frame of `main` there are `a = 3` and `b = 8` (Figure 1.4a).
2. **Call.** When `swap(&a, &b)` is called, a new frame is opened for `swap`. Addresses, not values, are copied into the parameters: `x` points to `a`; `y` points to `b`. `temp` has not yet received a value (Figure 1.4b).
3. `int temp = *x;` The value at the place `x` points to, that is 3, is read and written into `temp`.
4. `*x = *y;` The value `y` points to (8) is written to the place `x` points to, that is, to `a` in `main`. Now `a = 8` and `b = 8` (Figure 1.5a).
5. `*y = temp;` The 3 in `temp` is written to `b`, which `y` points to. The swap has taken place in the variables of `main` itself (Figure 1.5b).
6. **Return.** The frame of `swap` disappears; `x`, `y`, and `temp` no longer exist. But the change they made stays in the frame of `main`: `a = 8`, `b = 3` (Figure 1.5c).

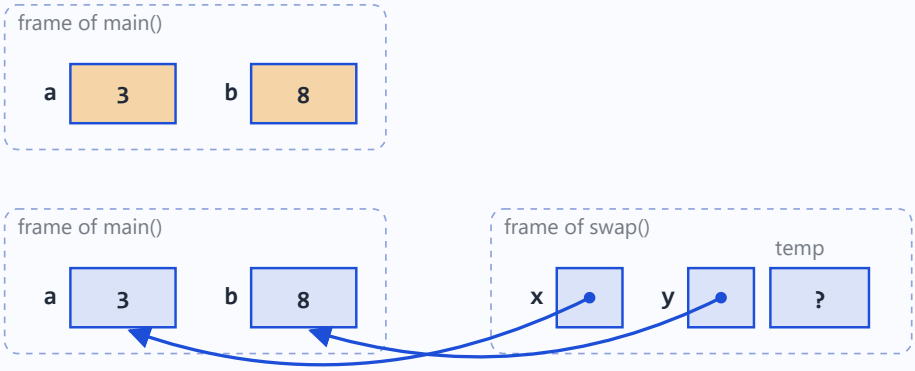
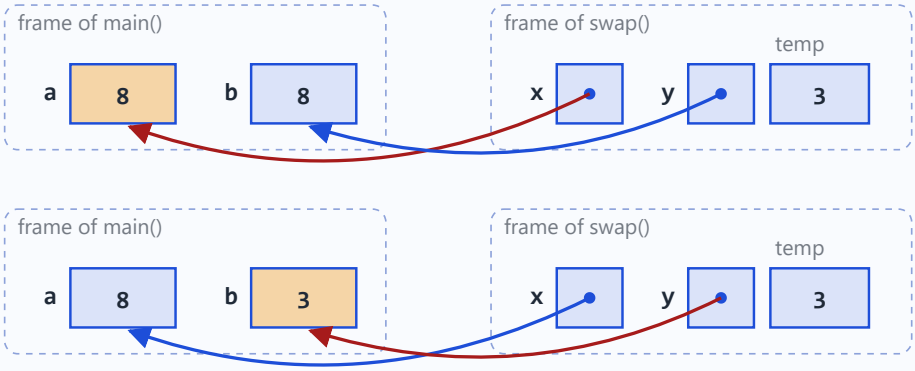


Figure 1.4: Memory before and just after the call `swap(&a, &b)`. (a) In the frame of `main`, `a = 3`, `b = 8`. (b) The frame of `swap` has been opened; `x` and `y` point to `a` and `b` respectively.



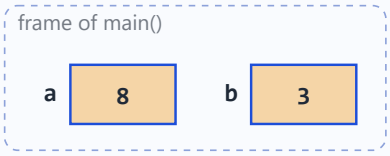


Figure 1.5: The last steps of the swap. (a) After `*x = *y`: `a` has become 8, and `temp` holds the old value (3). (b) After `*y = temp`: `b` has become 3. (c) `swap` has returned; its frame is gone, but `a = 8`, `b = 3` remain.

The same trace is given with variable values in Table 1.5. In the table the addresses are chosen as examples: 1000 for `a` and 1004 for `b`; an `int` occupies four bytes.

Step	`a` (1000)	`b` (1004)	`x`	`y`	`temp`
before the call	3	8	—	—	—
call	3	8	1000	1004	?
`temp = *x`	3	8	1000	1004	3
`*x = *y`	**8**	8	1000	1004	3
`*y = temp`	8	**3**	1000	1004	3
return	8	3	—	—	—

Table 1.5: Trace of the call `swap(&a, &b)` (addresses are examples).

Common mistake

Writing `swap(a, b)`, that is, sending values instead of addresses. Because the function expects `int *`, the compiler gives a warning or an error; if the warning is ignored, `x` points to address 3 and `y` to address 8, and `*x` accesses invalid memory. The program most likely crashes. Writing the function as `void swap(int x, int y)`, on the other hand, is a silent error: the program compiles and runs, but the values in `main` do not change.

1.6 Performance analysis

In computing the number of comparisons of selection sort, we found the cost of a program without running it, by looking only at the code. This section generalizes that approach. How much memory and time a program needs is computed as a function of the size of the input.

1.6.1 Evaluating a program

To decide whether a program is good, the following questions are asked:

1. Does it do what is required of it?

2. Does it work correctly?
3. Is it documented how to run and use it, comments included?
4. Does it use functions effectively to produce logical results?
5. Is the code readable?
6. Does it use primary and secondary memory effectively?
7. Is the running time acceptable?

The last two questions are the subject of performance evaluation. The first five are at least as important: a program that is fast but gives wrong results, is undocumented, and is unreadable soon loses its value because it cannot be maintained. But the answers to the first five questions are largely a matter of experience and care. The last two can be answered with numbers.

Performance evaluation is done in two ways.

- **Performance analysis** is *machine independent*. The program is not run; its time and memory requirements are estimated by looking at the code. The result is a complexity function.
- **Performance measurement** is *machine dependent*. The program is run on a particular machine with a particular compiler, and the real running time is measured.

The two ways complement each other. Analysis tells in advance how an algorithm will behave as the input grows. Measurement shows whether this prediction holds on a real machine, and brings out the constants that the analysis ignores.

1.6.2 Space complexity

Definition 1.3

The **space complexity** of a program P is the amount of memory the program needs to complete its work:

> $S(P) = c + S_P(I)$

Here c is the **fixed part**, independent of the size of the input and output; $S_P(I)$ is the **variable part**, which depends on the characteristics of the input I.

The fixed part covers the code of the program, simple variables, fixed-size structures, and constants. This memory is the same whatever the input. The variable part is the memory that changes with the input: arrays whose size is set by the input, the frames opened by recursive calls, blocks allocated by `malloc` according to the input. The characteristic of the input that affects this part most is usually its size: the number of elements of an array, the number of rows and columns of a matrix. In the analysis only the variable part is of interest, because it is what sets programs apart as the input grows.

Example 1.2

Consider this function, which computes an expression of three numbers:

```
float abc(float a, float b, float c) {
  return a + b + b * c
         + (a + b - c) / (a + b) + 4.00;
}
/* constant memory only: S_abc(I) = 0 */
```

The function takes three `float` parameters and returns a single value. There is no such thing as the size of the input; on every call the same three numbers and the same return address are stored. All the memory is in the fixed part: $S_{abc}(I) = 0$. The call `abc(1, 2, 3)` gives $1 + 2 + 2 \cdot 3 + (1 + 2 - 3)/(1 + 2) + 4 = 13$.

A habit about memory use should be acquired from the start: write code as if everything were limited. On today's computers memory looks plentiful, but the same code may run on an embedded system, on a phone, or on a server processing millions of records at once. Memory is not wasted just because there is plenty of it anyway.

1.6.3 The memory of sum and rsum

Let us compare two functions that add the elements of an array. `sum` uses a loop, and `rsum` uses recursion:

```
float sum(float list[], int n) {
  float tempsum = 0;
  int i;
  for (i = 0; i < n; i++)
    tempsum += list[i];
  return tempsum;
}

float rsum(float list[], int n) {
  if (n)
    return rsum(list, n - 1) + list[n - 1];
  return 0;
}
```

Recursion is a function calling itself with a smaller input. `rsum(list, n)` defines the sum of the first n elements as follows: if $n = 0$, the sum is 0; otherwise `list[n-1]` is added to the sum of the first $n - 1$ elements. The case $n = 0$, which gives a result directly without calling itself, is called the **base case**. Because every call reduces n by one, the chain eventually reaches the base case.

Let us find the variable memory of the two functions.

- To `sum`, not the elements of the array but **the address of its first element** is passed. In C an array parameter is really a pointer; `float list[]` and `float *list` mean the same thing as a parameter. Therefore, however large the array is, `sum` stores only one address, one `int`, and two local variables: $S_{sum}(n) = 0$. Had the elements been copied by value, as when an array is passed as a value parameter in Pascal, it would have been $S_{sum}(n) = n$.
- `rsum(list, n)` is called $n + 1$ times in all: once each for the values $n, n - 1, \ldots, 1, 0$. Each call opens a frame on the stack, and these frames live together until the base case is reached. Therefore the variable memory is proportional to n: $S_{rsum}(n) \propto n$.

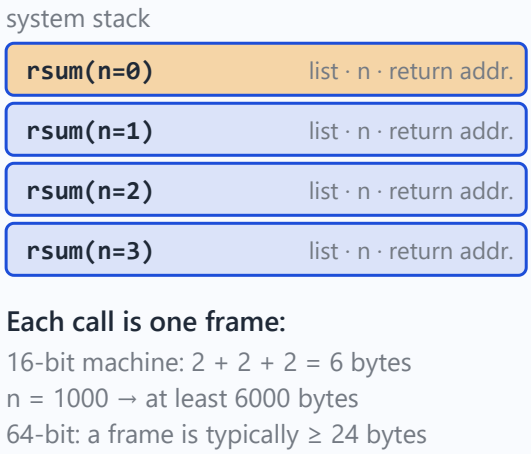


Figure 1.6: The system stack at the deepest moment of the call `rsum(list, 3)`. Four frames sit on top of one another; each stores the pointer `list`, `n`, and the return address.

Figure 1.6 shows the contents of one frame: the address of the array (`list`), the parameter `n`, and the return address that says where the program continues when the function ends. On the 16-bit machine assumed by the textbook, each of these three values is 2 bytes and a frame takes 6 bytes. For $n = 1000$, 1001 frames, that is at least 6006 bytes, are needed. On today's 64-bit machines addresses are 8 bytes, and compilers also add alignment and register-saving space to the frame. A frame is typically at least 24 bytes, often more.

This difference has a practical consequence. The memory set aside for the call stack is limited; on Windows the default is around 1 MB. If `rsum` is called with a very large n, the frames exceed this limit and the program crashes with a **stack overflow**. In a test made while preparing these notes, `rsum` gave the correct result for $n = 100{,}000$, while for $n = 1{,}000{,}000$ the program crashed. With the same input `sum` ran without trouble. The exact place of the limit varies with the compiler, the operating system, and the build options.

1.6.4 Time complexity and the program step

The time *T(P)* a program *P* takes is the sum of the compile time and the run time. Compilation is done once, and the program can be run many times afterwards. So only the run time is of interest; this time is denoted t_P.

The run time can be written as the sum of the products of the number of each basic operation the program performs and the time of that operation:

> $t_P(n) = c_a \cdot ADD(n) + c_s \cdot SUB(n) + c_l \cdot LDA(n) + c_{st} \cdot STA(n) + \ldots$

Here *ADD(n)* is the number of additions performed for an input of size *n*, and c_a is the time of one addition. There are terms for subtraction (SUB), loading from memory (LDA), and storing to memory (STA) as well. The formula is exact but impractical. The constants *c* depend on the compiler, the processor, and the state of the cache, and are hard to know in advance. It is also not clear how many machine instructions a compiler will translate a C statement into.

For this reason time is measured in **program steps** instead of machine instructions.

> **Definition 1.4**
>
> A **program step** is a syntactically or semantically meaningful program segment whose running time is independent of the size of the input. The **time complexity** of a program is the number of program steps it executes as a function of the size of the input.

The crucial point of the definition is the phrase "independent of the size of the input". An assignment such as `a = b + c * d;` counts as a single step; even though it turns into several machine instructions, its time does not depend on *n*. A loop cannot be counted as a single step, because how many times it runs depends on *n*. Each test of the loop is a step, and each execution of each statement in its body is a separate step. Declarations (such as `int i;`) do no work at run time and so are not counted as steps. The function header and the closing brace are not steps either.

The step count is machine independent, yet it keeps its relation to real time: if the step count doubles, the time also roughly doubles. Although the time of a step varies from machine to machine, this ratio does not.

An extreme example shows why performance matters so much. Suppose the bombing system of a fighter plane is being programmed. If this program finds the target and drops the bomb in 3 seconds, and the opponent's does the same job in 2 seconds, the war has been lost from the start. In real-time systems, in servers processing big data, and in games, a difference of one second is the difference between a useful program and a useless one.

1.6.5 Counting steps: sum()

An orderly way to find the step count of a function is the **step-count table**. In the table, three numbers are written for each statement. **s/e** (steps per execution) is how many steps one execution of the statement takes. **Frequency** is how many times the statement is executed. **Total** is the product of the two. The step count of the function is the sum of the total column. The table for the function `sum` is given in Table 1.6.

Statement	s/e	Frequency	Total
`float sum(float list[], int n) {`	0	0	0
`float tempsum = 0;`	1	1	1
`int i;`	0	0	0
`for (i = 0; i < n; i++)`	1	*n + 1*	*n + 1*
`tempsum += list[i];`	1	*n*	*n*
`return tempsum;`	1	1	1
`}`	0	0	0
Total			*2n + 3*

Table 1.6: Step-count table of the function `sum`.

The row of the table that needs the most care is the `for` row. The loop body is executed *n* times, but the loop condition is tested *n + 1* times. For the values *i = 0, 1, ..., n − 1* the test is true and the body is entered; these account for *n* tests. When *i = n*, the test is made once more, comes out false, and the loop ends. This last test is also a step. The assignment line and the `return` are executed once each. The total is *1 + (n + 1) + n + 1 = 2n + 3* steps.

1.6.6 Verifying with a counter

The correctness of the table can be checked by adding a counter to the program. A global variable `count` starts at zero and is increased by one at every program step:

```
float sum(float list[], int n) {
  float tempsum = 0; count++;
  int i;
  for (i = 0; i < n; i++) {
    count++;              /* for test */
    tempsum += list[i]; count++;
  }
  count++;  /* last test of for */
  count++;  return tempsum;
}
```

The counter increments stand in place of the steps in the table: one after the assignment, one at each loop test, one at each execution of the body, one at the last test of the loop, and one at the `return`. The counter for the loop test is placed at the start of the body, because a test has been made before each entry into the body. To count the exit test, there is a separate increment after the loop.

For `list = {3, 1, 4}` and `n = 3`, the trace is given in Table 1.7.

Event	`i`	`tempsum`	`count`
start	—	—	0
`tempsum = 0`	—	0	1
test 0 < 3: true	0	0	2
`tempsum += list[0]`	0	3	3
test 1 < 3: true	1	3	4
`tempsum += list[1]`	1	4	5
test 2 < 3: true	2	4	6
`tempsum += list[2]`	2	8	7
test 3 < 3: false	3	8	8
`return tempsum`	3	8	9

Table 1.7: Trace of the call `sum(list, 3)` with the counter; `list = {3, 1, 4}`.

In the end `count = 9 = 2 · 3 + 3`; the formula in the table is confirmed. The function returns *3 + 1 + 4 = 8*. The function with the counter was also run for *n = 0, 1, ..., 6*, and every time the value of `count` came out equal to *2n + 3*.

The code with the counter has one detail worth noticing: after the line `float tempsum = 0; count++;` comes the declaration `int i;`. Declaring in the middle of a block was forbidden in C89 and has been allowed since C99. The code must therefore be compiled with C99 or a newer standard.

1.6.7 Counting steps: rsum()

The step count of a recursive function is found a little differently: the frequency of each statement is computed from how many times the function is called. `rsum(list, n)` is called *n + 1* times in all. The step-count table is given in Table 1.8.

Statement	s/e	Frequency	Total
`float rsum(float list[], int n) {`	0	0	0
`if (n)`	1	*n + 1*	*n + 1*
`return rsum(list, n-1) + list[n-1];`	1	*n*	*n*
`return 0;`	1	1	1
`}`	0	0	0

Statement	s/e	Frequency	Total
Total			*2n + 2*

Table 1.8: Step-count table of the function `rsum`.

The test `if (n)` is made once per call, that is, *n + 1* times. In the *n* calls made while *n* is nonzero, the second line is executed: the result of the recursive call is added to an element and returned. This line counts as one step; the steps of the called function itself are counted separately, in the lines of that call. In the base case `return 0` is executed once. The total is *(n + 1) + n + 1 = 2n + 2*.

At first glance this result makes `rsum` look faster than `sum` (*2n + 3*). But the step count is misleading here. The time of a step is independent of the input, but not all steps take the same time. A recursive call requires opening a new frame, copying the parameters, saving the return address, and removing the frame on return. This work is much more expensive than a comparison and an increment in a loop iteration. On a real machine `rsum` is usually slower than `sum` and uses extra memory.

What the step count really tells is something else: how the step count grows as *n* grows. *2n + 3* and *2n + 2* grow in direct proportion to *n*; both are **linear** and both are *O(n)*. When *n* doubles, the time of both functions roughly doubles.

1.6.8 rsum() and the call stack

A counter can be added to `rsum` as well:

```
float rsum(float list[], int n) {
  count++;                /* if test */
  if (n) {
    count++;              /* addition + return */
    return rsum(list, n - 1) + list[n - 1];
  }
  count++;
  return 0;
}
```

The trace of the call `rsum(list, 3)`, together with the state of the call stack at every step, is given in Table 1.9. In the stack column, the rightmost call is the one running at that moment.

Event	Call stack (bottom to top)	`count`
`rsum(list, 3)`: `if` test, 3 ≠ 0	r(3)	1
addition and `return` step; `rsum(list, 2)` is needed first	r(3)	2
`rsum(list, 2)`: `if` test, 2 ≠ 0	r(3), r(2)	3
`rsum(list, 1)` is needed first	r(3), r(2)	4
`rsum(list, 1)`: `if` test, 1 ≠ 0	r(3), r(2), r(1)	5
`rsum(list, 0)` is needed first	r(3), r(2), r(1)	6

Event	Call stack (bottom to top)	`count`
`rsum(list, 0)` : `if` test, *n = 0*: base case	r(3), r(2), r(1), r(0)	7
`return 0` ; the stack is at its deepest	r(3), r(2), r(1), r(0)	8
0 + `list[0]` = 3; `rsum(list, 1)` returns 3	r(3), r(2), r(1)	8
3 + `list[1]` = 4; `rsum(list, 2)` returns 4	r(3), r(2)	8
4 + `list[2]` = 8; `rsum(list, 3)` returns 8	r(3)	8

Table 1.9: Trace of the call `rsum(list, 3)` with the counter; `list = {3, 1, 4}` .

Two phases are clearly visible in the trace. In the **descent phase** each call makes the `if` test, increments `count` once more, and immediately opens a new call. The addition cannot be done yet, because the result of the recursive call, which is the first term of the sum, is not yet known. The stack grows to four frames. In the **return phase**, starting from the base case, each call adds its own element to the value it receives and returns to the call below. The additions are done in this phase, starting from `list[0]` . The counter does not increase in the return phase, because the addition and `return` steps were counted in the descent phase. The final value is *count = 8 = 2 · 3 + 2*.

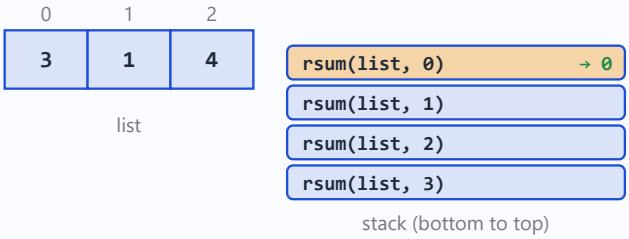


Figure 1.7: The deepest moment of the trace of `rsum(list, 3)` . There are four frames on the stack; the topmost, `rsum(list, 0)` , has reached the base case and is about to return 0.

1.6.9 Matrix addition

The step count of nested loops is found the same way. The table for the function `add` , which adds two matrices, is in Table 1.10. The function performs `c[i][j] = a[i][j] + b[i][j]` for every cell.

Statement	Frequency
`void add(int a[][MAX_SIZE], int b[][MAX_SIZE],` `  int c[][MAX_SIZE], int rows, int cols) {`	0
`int i, j;`	0
`for (i = 0; i < rows; i++)`	*rows + 1*
`  for (j = 0; j < cols; j++)`	*rows · (cols + 1)*
`  c[i][j] = a[i][j] + b[i][j];`	*rows · cols*
Total	*2·rows·cols + 2·rows + 1*

Table 1.10: Step-count table of the matrix addition function `add` (s/e = 1 for every statement).

The test of the outer loop, as in `sum`, is made *rows + 1* times. The inner loop starts afresh on every iteration of the outer loop; each time it starts, its test is made *cols + 1* times. Since the outer loop runs *rows* iterations, the total frequency of the inner loop test is *rows · (cols + 1)*. The assignment is executed once for every cell, *rows · cols* times in total. Adding everything up:

> $(rows + 1) + rows \cdot (cols + 1) + rows \cdot cols = 2 \cdot rows \cdot cols + 2 \cdot rows + 1$

For example, for *rows = 2* and *cols = 4* one gets *16 + 4 + 1 = 21* steps. The formula was tested by adding a counter to the function, for four different row counts and two different column counts; in all of them the counter gave the same value as the formula.

The growth rate of the result is $O(rows \cdot cols)$; for a square matrix ($rows = cols = n$) this is $O(n^2)$. The term *2·rows* in the formula depends on which dimension the outer loop runs over. If the number of rows is much larger than the number of columns ($rows \gg cols$), the loops can be interchanged: the outer loop runs over columns and the inner loop over rows. The count then becomes *2·rows·cols + 2·cols + 1*, and the small term shrinks. The dominant term does not change. Matrix multiplication, in contrast, requires three nested loops and is $O(n^3)$.

Common mistake

Writing the complexity of matrix addition as "O(rows, cols)". A function goes inside the O notation; separating two dimensions with a comma carries no meaning. Because the number of operations grows with the product of the two dimensions, the correct form is $O(rows \cdot cols)$.

1.6.10 Best, average, and worst case

In the functions examined so far, the step count depended only on the size of the input: `sum` takes *2n + 3* steps on every array of *n* elements. But in many algorithms two inputs of the same size require different numbers of steps. **Linear search** is the simplest example: a value is compared one by one with the elements of the array from the beginning to the end, and the search stops when it is found.

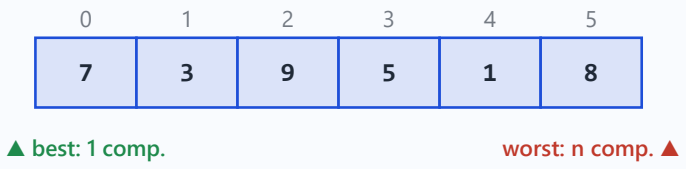


Figure 1.8: Linear search in an array of six elements. If the sought value is in the first cell (7), a single comparison suffices; if it is in the last cell (8) or is not in the array, *n* comparisons are needed.

In the array {7, 3, 9, 5, 1, 8} of Figure 1.8, if 7 is searched for, it is found at the first comparison. If 8 is searched for, six comparisons are needed. If 6 is searched for, it is established after six comparisons that it is not there. These three cases are given names:

Case	When?	Comparisons
Best	The sought value is in the first cell	$1 \to \Theta(1)$
Average	The sought value is in the array and is equally likely to be at any position	$(n + 1) / 2 \to \Theta(n)$
Worst	The sought value is in the last cell or not in the array at all	$n \to \Theta(n)$

Table 1.11: Best, average, and worst case of linear search in an array of n elements.

- The **best case** is the cost of the input that requires the fewest steps among inputs of the same size.
- The **worst case** is the cost of the input that requires the most steps.
- The **average case** is the expected cost under a given probability distribution of the inputs.

The average is found as follows. Let the sought value be in the array, and let the probability of finding it at each of the n positions be $1/n$. If the value is at position k (counting from 1), k comparisons are made. The expected number of comparisons is:

> $(1/n) \cdot (1 + 2 + \ldots + n) = (1/n) \cdot n(n + 1)/2 = (n + 1)/2$

For an array of six elements this is 3.5: on average half of the array is scanned. Even though it is half, the scan is directly proportional to n; so the average case is also $\Theta(n)$. The notation $\Theta(n)$ says that the step count grows at the same rate as n, that is, it stays between two constant multiples of n.

The analysis usually gives the worst case. There are two reasons. First, the worst case is a guarantee valid for every input: the algorithm never takes more steps than this on any input. In a real-time system, or in an operation a user is waiting for, this guarantee is worth more than the average. Second, computing the average case requires knowing the probability distribution of the inputs. This distribution is often unknown, and an assumed distribution may not match reality. The best case, for its part, rarely gives useful information: almost every algorithm is fast on some lucky inputs.

1.7 Asymptotic notation

Finding the step count exactly is not always easy, and often it is not necessary. It is nice to know that `sum` takes $2n + 3$ steps, but whether the number is $2n + 3$ or $3n + 2$ hardly matters. The real time of a step is unknown anyway; constant factors in the step count get lost in that uncertainty. What matters is *how* the step count grows as n grows: linearly, quadratically, exponentially? The language that answers this question stripped of constants and small terms is called **asymptotic notation**.

1.7.1 Big O

Definition 1.5

If there exist positive constants c and n_0 such that $f(n) \le c \cdot g(n)$ for all $n \ge n_0$, one writes $f(n) = O(g(n))$, read "$f(n)$ is of order $g(n)$". This notation is called **Big-O**.

The definition allows two freedoms. The first is the constant c: not $g(n)$ itself but some constant multiple of it must stay above $f(n)$. This freedom makes constant factors irrelevant. The second is the constant n_0: the inequality is required only for sufficiently large n. What happens at small n is ignored. Figure 1.9 shows this situation: before n_0 the curve $f(n)$ may lie above $c \cdot g(n)$, but after n_0 it always stays below.

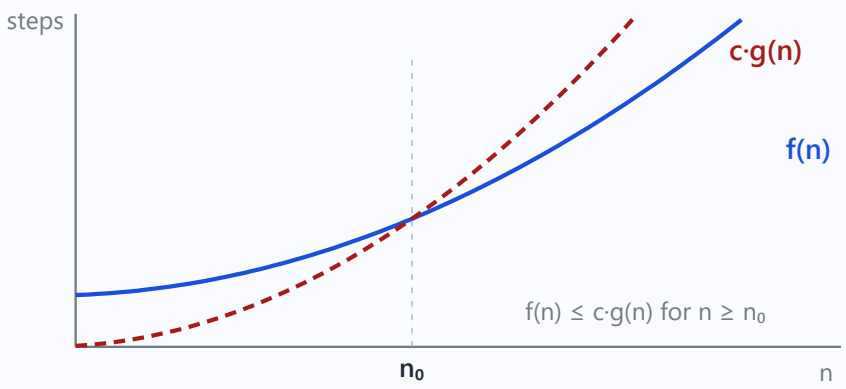


Figure 1.9: The definition of Big-O. After n_0 the curve $f(n)$ stays below the curve $c \cdot g(n)$.

The step counts found for `sum`, `rsum`, and `add` are written in this notation as follows:

- $T_{sum}(n) = 2n + 3 = O(n)$. Since $2n + 3 \le 3n$ for $n \ge 3$, one can choose $c = 3$, $n_0 = 3$.
- $T_{rsum}(n) = 2n + 2 = O(n)$. For $n \ge 2$, $2n + 2 \le 3n$.
- $T_{add} = 2{\cdot}rows{\cdot}cols + 2{\cdot}rows + 1 = O(rows \cdot cols)$; $O(n^2)$ for a square matrix.

The "=" sign in the notation is not the usual equality. $O(g(n))$ is the set of all functions bounded from above by a constant multiple of $g(n)$; the statement $f(n) = O(g(n))$ means "$f(n)$ is an element of this set". So the statement is read in one direction only: one writes $2n + 3 = O(n)$, but not $O(n) = 2n + 3$. Big-O is an **upper bound**; it says how fast the function grows at most.

In this course time complexity is computed most of the time. Memory matters too, but in most problems time is the scarcer resource: missing memory can sometimes be added, while time that has passed does not come back.

1.7.2 Example: $3n^3 + 6n^2 + 7n$

Example 1.3

Let $f(n) = 3n^3 + 6n^2 + 7n$. Let us show that $f(n) = O(n^3)$ by finding c and n_0 that fit the definition. The method is to bound the nondominant terms step by step by the next higher power.

1. The smallest term is $7n$. For $n \geq 7$, $7n \leq n \cdot n = n^2$. Hence $f(n) \leq 3n^3 + 6n^2 + n^2$.
2. The squared terms combine: $6n^2 + n^2 = 7n^2$. Now $f(n) \leq 3n^3 + 7n^2$.
3. The same route once more: for $n \geq 7$, $7n^2 \leq n \cdot n^2 = n^3$. Thus $f(n) \leq 3n^3 + n^3 = 4n^3$.
4. We have found $c = 4$ and $n_0 = 7$: for all $n \geq 7$, $f(n) \leq 4n^3$. The definition is satisfied.

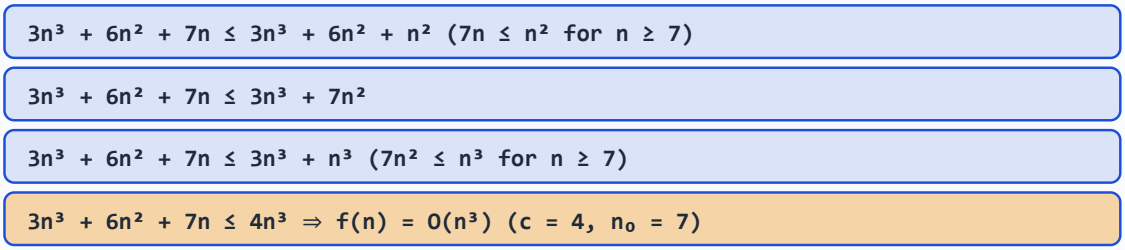


Figure 1.10: The chain of inequalities $3n^3 + 6n^2 + 7n \leq 4n^3$. In each line a nondominant term is raised to the next higher power under the condition $n \geq 7$.

The c and n_0 found are not the only choice. For example, for $n \geq 1$ every term can be bounded by a multiple of n^3: $f(n) \leq 3n^3 + 6n^3 + 7n^3 = 16n^3$. This gives $c = 16$, $n_0 = 1$. The definition requires only that *some* constants exist; which ones are chosen does not matter. For $c = 4$, $n_0 = 7$ is indeed the smallest value: when tried with a script, at $n = 6$ one gets $f(6) = 906 > 4 \cdot 216 = 864$.

The result has two practical readings.

- **Upper bound.** If the running time is given by $f(n)$ steps, then for $n \geq 7$ the program never exceeds $4n^3$ steps.
- **Cubic growth.** When n triples, the time grows by at most $3^3 = 27$ times. If this algorithm sorts 100 numbers in 2 seconds, sorting 300 numbers takes at most $2 \cdot 27 = 54$ seconds; not 6 seconds.

The method in the example can be generalized.

Theorem 1.1 (Degree of a polynomial)

If $f(n) = a_m n^m + a_{m-1} n^{m-1} + \ldots + a_1 n + a_0$ is a polynomial, then $f(n) = O(n^m)$.

Proof. Let $n \geq 1$. For every $i \leq m$, $n^i \leq n^m$. Let us bound with the absolute values of the coefficients: $f(n) \leq |a_m|n^m + \ldots + |a_1|n + |a_0| \leq (|a_m| + \ldots + |a_1| + |a_0|) \cdot n^m$. If $c = |a_m| + \ldots + |a_0|$ and $n_0 = 1$ are chosen, the definition is satisfied. □

The theorem says that the growth rate of a polynomial is determined by its highest-degree term. This is why $2n + 3$, $100n$, and $n + 10^6$ are in the same class: all are $O(n)$.

1.7.3 Ω and Θ

Big-O gives only an upper bound, and this bound can be loose. The statement $3n + 2 = O(n^2)$ is true: for $n \geq 2$, $3n + 2 \leq 4n \leq 4n^2$. Even $3n + 2 = O(2^n)$ is true. But these statements tell little about the function. To state the growth rate of a function exactly, the notions of lower bound and tight bound are also needed.

Definition 1.6

If there exist positive c and n_0 such that $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$, one writes $f(n) = \Omega(g(n))$ (**Big Omega**). If there exist positive c_1, c_2, and n_0 such that $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all $n \geq n_0$, one writes $f(n) = \Theta(g(n))$ (**Big Theta**).

Notation	Definition	Meaning
$f(n) = O(g(n))$	$f(n) \leq c \cdot g(n)$	g is an upper bound of f
$f(n) = \Omega(g(n))$	$f(n) \geq c \cdot g(n)$	g is a lower bound of f
$f(n) = \Theta(g(n))$	$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$	both O and Ω: tight bound

Table 1.12: The three asymptotic notations (inequalities for all $n \geq n_0$).

Θ is the intersection of O and Ω: $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. Θ says that the function grows *at the same rate* as $g(n)$.

Example 1.4

Let us show that $3n + 2 = \Theta(n)$. For the lower bound: for every $n \geq 1$, $3n + 2 \geq 3n$; that is, $c_1 = 3$. For the upper bound: if $n \geq 2$ then $2 \leq n$ and $3n + 2 \leq 3n + n = 4n$; that is, $c_2 = 4$. Taking $n_0 = 2$, $3n \leq 3n + 2 \leq 4n$ holds for all $n \geq 2$. At $n = 1$ the upper bound fails ($5 > 4$); this is why n_0 is chosen as 2.

When using asymptotic notation, the smallest possible upper bound is written. $3n + 2 = O(n^2)$ is true but misleading; a reader may think the algorithm is quadratic. If a tight bound is known, it is written with Θ. Selection sort makes $n(n - 1)/2$ comparisons whatever the input. Since $n^2/4 \leq n(n - 1)/2 \leq n^2/2$ for $n \geq 2$, selection sort is $\Theta(n^2)$. In linear search, the worst case is $\Theta(n)$ and the best case is $\Theta(1)$; if a single bound valid for every input is wanted, one writes $O(n)$.

These notations also make it easier to think when a problem can be solved better than the required bound. Suppose an exam question asks "solve it in $O(n + m)$ time". A student has solved the problem in $O(n)$, then, to follow the wording of the question, added an empty loop running m times at the end of the code. This is unnecessary. A solution that takes $O(n)$ time is already $O(n + m)$, because Big-O is an upper bound. If something better than the required bound can be done, so much the better.

1.7.4 Common complexities

The time complexities of algorithms mostly fall into one of a few familiar classes. Table 1.13 gives these classes, their names, and the values they take for small n. In the table $lg\ n$ means $log_2\ n$.

Complexity	Name	n=1	n=2	n=4	n=8	n=16	n=32
1	constant	1	1	1	1	1	1
$lg\ n$	logarithmic	0	1	2	3	4	5
n	linear	1	2	4	8	16	32
$n\ lg\ n$	linearithmic	0	2	8	24	64	160
n^2	quadratic	1	4	16	64	256	1,024
n^3	cubic	1	8	64	512	4,096	32,768
2^n	exponential	2	4	16	256	65,536	$4.29 \cdot 10^9$
$n!$	factorial	1	2	24	40,320	$2.09 \cdot 10^{13}$	$2.63 \cdot 10^{35}$

Table 1.13: Magnitude of the common complexity functions at small values of n.

Each class has a typical source. A **constant**-time operation, such as accessing an element of an array by index, does not look at the size of the input. **Logarithmic** time is seen in algorithms that halve the problem at each step; binary search in a sorted array is an example. **Linear** time is the time of algorithms that look at the input once, for example `sum` and linear search. **Linearithmic** time appears in good sorting algorithms. **Quadratic** time comes from two nested loops and **cubic** time from three nested loops; selection sort and matrix multiplication are examples. **Exponential** and **factorial** times are the times of algorithms that try all subsets or all orderings.

The small values of n in the table can be misleading. At $n = 4$, $2^n = n^2 = 16$; the two functions look the same. At $n = 8$ the exponential function has pulled ahead: $2^n = 256$, $n^2 = 64$. At $n = 32$ the difference is expressed in billions. Beyond a certain point exponential and factorial terms crush all the polynomial terms.

1.7.5 Growth graph

A graph of the same functions between $n = 1$ and $n = 12$ shows the difference even more clearly. In the upper graph of Figure 1.11 the vertical scale runs from 0 to 150. On this scale $lg\ n$, n, and $n\ lg\ n$ separate from one another; n^2 climbs quickly and reaches 144 at $n = 12$. n^3 and 2^n have long since left the graph.

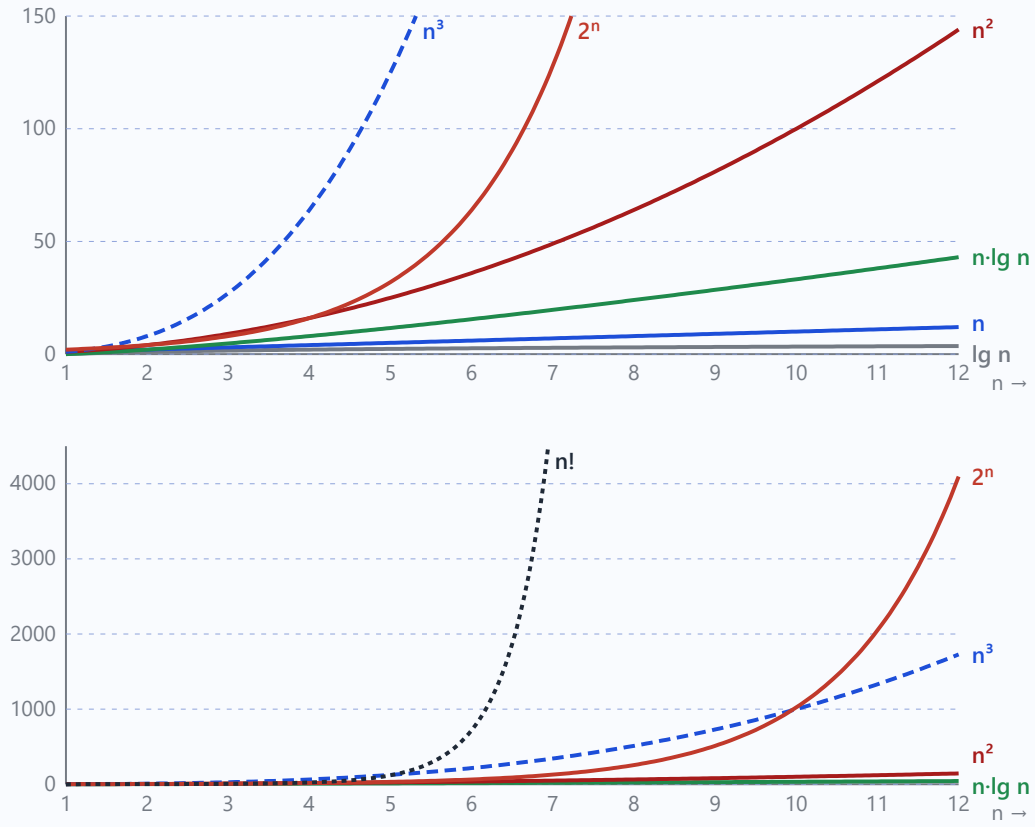


Figure 1.11: Growth of the complexity functions for *n = 1 ... 12*. In the top graph the vertical scale is 0–150, in the bottom 0–4500. In the bottom graph *n!* leaves the graph at *n = 7*; the linear functions look glued to the floor.

In the lower graph the scale is widened to 0 to 4500. At $n = 12$, $n^3 = 1728$ and $2^n = 4096$. 2^n passes n^3 around $n = 10$ and stays ahead from then on. $n!$ leaves the graph at $n = 7$ with the value 5040; at $n = 6$ it is only 720. On this scale n and $n \lg n$ cannot be told apart from the horizontal axis.

1.7.6 A billion instructions per second

To see the effect of complexity on real time, let us imagine a machine that executes 10^9 instructions per second, that is, performs each step in 1 nanosecond. Table 1.14 shows how long an algorithm that takes $f(n)$ steps would take on this machine. In the computation 1 year is taken as 365 days and 1 month as 30 days. All the values in the table were recomputed and verified with a script.

n	n	lg n	n lg n	n^2	n^3	n^4	n^{10}	2^n
10	0.01 µs	0.00332 µs	0.0332 µs	0.1 µs	1 µs	10.0 µs	10.0 s	1.02 µs
20	0.02 µs	0.00432 µs	0.0864 µs	0.4 µs	8 µs	160 µs	2.84 hours	1.05 ms
30	0.03 µs	0.00491 µs	0.147 µs	0.9 µs	27.0 µs	810 µs	6.83 days	1.07 s

n	n	lg n	n lg n	n^2	n^3	n^4	n^{10}	2^n
40	0.04 µs	0.00532 µs	0.213 µs	1.6 µs	64.0 µs	2.56 ms	4.05 months	18.3 min
50	0.05 µs	0.00564 µs	0.282 µs	2.5 µs	125 µs	6.25 ms	3.1 years	13.0 days
100	0.1 µs	0.00664 µs	0.664 µs	10.0 µs	1 ms	100 ms	31.7 centuries	$4.02 \cdot 10^{11}$ centuries
1000	1 µs	0.00997 µs	9.97 µs	1 ms	1 s	16.7 min	$3.17 \cdot 10^{11}$ centuries	$3.40 \cdot 10^{282}$ centuries
10000	10.0 µs	0.0133 µs	133 µs	100 ms	16.7 min	3.86 months	$3.17 \cdot 10^{21}$ centuries	$6.33 \cdot 10^{2991}$ centuries

Table 1.14: Time for *f(n)* steps on a machine executing 10^9 instructions per second.

To see how a cell is computed, take $n = 40$ and $f(n) = 2^n$. $2^{40} \approx 1.1 \cdot 10^{12}$ steps is $1.1 \cdot 10^{12}$ ns, that is, about 1100 seconds. That makes 18.3 minutes. When n grows by 10 to 50, the step count grows by a factor of $2^{10} = 1024$ and the time reaches 13 days. In exponential algorithms, adding a few elements to the input multiplies the time.

Three conclusions follow from the table. First, logarithmic and linear algorithms finish within microseconds even on ten thousand elements. Second, there are large differences among polynomial algorithms too: on ten thousand elements n^2 takes 100 milliseconds, n^3 takes 16.7 minutes, and n^4 takes 3.86 months. Third, exponential algorithms are usable in practice only on very small inputs. For $n = 100$, 2^n steps take $4 \cdot 10^{11}$ centuries; the age of the universe is about $1.4 \cdot 10^8$ centuries. For $n = 10{,}000$ the time is about 10^{2992} centuries.

1.7.7 Amortized analysis

Worst-case analysis treats each operation on its own. But in some data structures operations are sometimes expensive and mostly cheap. In such a structure, adding up the worst-case cost of every operation makes the total cost look far larger than it is. **Amortized analysis** computes the total cost of a sequence of n operations and divides it by n. The result is the average cost per operation. This average does not rest on a probabilistic assumption; it is a guarantee valid for every sequence of operations.

The typical example is the **dynamic array**. A dynamic array starts with a certain **capacity** and appends elements at the end. When the array becomes full, its capacity is doubled with `realloc`. `realloc` enlarges an allocated block; if it cannot enlarge it in place, it allocates a new block and copies the old elements there. The cost of an insertion is one write plus, if any, the number of elements copied.

Example 1.5

Let 9 elements be appended to an empty array of capacity 1, with the capacity doubling each time it fills. At the 2nd insertion the array is full: 1 element is copied and the capacity becomes 2. At the 3rd insertion 2 elements are copied and the capacity becomes 4. At the 5th insertion 4 elements are copied and the capacity becomes 8. At the 9th insertion 8 elements are copied and the capacity becomes 16. The total number of copies is $1 + 2 + 4 + 8 = 15$, and the total number of writes is 9. The total cost of 9 insertions is 24; this comes to $24 / 9 \approx 2.7$ operations per insertion. Looked at individually, the 9th insertion cost 9 operations, but the average stays below 3.

In the general case, the number of elements copied in n insertions is $1 + 2 + 4 + \ldots + 2^k$, where $2^k < n$. This geometric sum is $2^{k+1} - 1 < 2n$. Together with the writes, the total cost is less than $3n$. A single insertion may take $O(n)$, but the **amortized cost** per insertion is $O(1)$. This bound was also checked with a script for different values of n between 1 and 1000; the total cost stayed below $3n$ every time.

Had the capacity been increased by a fixed amount, say 10, each time instead of being doubled, the result would be very different. In that case the whole array is copied once every 10 insertions, and the number of copies is $10 + 20 + 30 + \ldots \approx n^2/20$. The amortized cost per insertion rises to $O(n)$. Growing by doubling is the idea that dynamic arrays and many library structures rest on.

1.8 The magic square

A problem that shows the practical importance of asymptotic notation strikingly is the magic square. When two methods for the same problem are compared, one working by brute force and the other by a clever rule, one sees that the right algorithm makes a difference that no hardware speedup can make up for.

1.8.1 The problem

Definition 1.7

A **magic square** of size $n \times n$ is a table in which the numbers from 1 to n^2 are placed so that the sums of every row, every column, and the two main diagonals are equal.

This common sum can be computed in terms of n. The sum of the numbers from 1 to n^2 is $n^2(n^2 + 1)/2$. These numbers are spread over n rows, and every row has the same sum. So the sum of one row is:

$$n^2(n^2 + 1)/2 \div n = n(n^2 + 1)/2$$

For $n = 3$ the sum is $3 \cdot 10 / 2 = 15$, for $n = 5$ it is $5 \cdot 26 / 2 = 65$, and for $n = 7$ it is $7 \cdot 50 / 2 = 175$. Figure 1.12 shows a magic square of size 5 × 5.

15	8	1	24	17
16	14	7	5	23
22	20	13	6	4
3	21	19	12	10
9	2	25	18	11

Figure 1.12: A 5 × 5 magic square. The sum of every row, column, and main diagonal is 65. The square was generated with Coxeter's rule: the 1 is in the middle of the first row.

Let us check the square in the figure. The first row is $15 + 8 + 1 + 24 + 17 = 65$, the first column $15 + 16 + 22 + 3 + 9 = 65$, the main diagonal $15 + 14 + 13 + 12 + 11 = 65$, the other diagonal $17 + 5 + 13 + 21 + 9 = 65$. The remaining rows and columns also add up to 65.

1.8.2 Brute force versus a faster machine

The first way that comes to mind for solving the problem is **brute force**: trying all possible solutions one by one. The numbers from 1 to n^2 can be arranged in n^2 cells in $(n^2)!$ different ways. For each arrangement the row, column, and diagonal sums are checked. For a 5 × 5 square there are $25! \approx 1.55 \cdot 10^{25}$ arrangements. A machine that tries 10^9 arrangements per second tries all of them in $1.55 \cdot 10^{16}$ seconds. Since a century is about $3.15 \cdot 10^9$ seconds, this time is about 4.9 million centuries.

A common objection to this computation is that a computer ten times faster, say a new 10 GHz one, would solve this problem at once, so there is no need to bother with the algorithm. The computation refutes this objection:

> $25! / (10^{10} \cdot 3.15 \cdot 10^9) \approx 4.9 \cdot 10^5$ centuries

A machine ten times faster brings 4.9 million centuries down to about 490 thousand centuries. The result is still hopeless. Even a machine a thousand times faster brings the time down only to 4900 centuries. And in an algorithm of complexity $(n^2)!$, raising n from 5 to 6 multiplies the step count by $36!/25! \approx 2.4 \cdot 10^{16}$; no hardware progress can cover that.

The right algorithm, on the other hand, builds the same square in only 25 steps, in far less than a microsecond. The algorithm comes before the hardware: running a slow algorithm on a fast machine is almost always worse than running a fast algorithm on a slow machine.

1.8.3 Coxeter's rule and the C code

When n is odd, there is a simple rule that builds the magic square directly. Horowitz and Sahni attribute this rule to Coxeter; it is also known as the Siamese method or the de la Loubère method. The rule is this:

1. Write 1 in the middle of the first row.
2. For the next number always move to the upper-left diagonal: one row up, one column left.
3. If you run off the table, wrap around from the opposite edge. A move that goes off the top goes to the bottom row, and a move that goes off the left goes to the rightmost column. The table is thought of as a torus whose opposite edges are glued together.
4. If the upper-left diagonal cell is occupied, do not go there; instead step down to the cell directly below the current one.

The rule places each number in a single step and finishes in n^2 steps: the time complexity is $O(n^2)$. Since the table itself already contains n^2 cells, there cannot be a faster algorithm; every cell must be written at least once. The rule is therefore an algorithm with $\Theta(n^2)$ time that cannot be bettered. The rule written in C is this:

```c
void magic(int n, int sq[][MAX]) {
  int i = 0, j = (n - 1) / 2, s, r, c;
  sq[i][j] = 1;       /* sq must be zeroed */
  for (s = 2; s <= n * n; s++) {
    r = (i - 1 < 0) ? n - 1 : i - 1; /* up */
    c = (j - 1 < 0) ? n - 1 : j - 1; /* left */
    if (sq[r][c]) i = (i + 1) % n; /* down */
    else { i = r; j = c; }
    sq[i][j] = s;
  }
}
```

The code works as follows, line by line:

- `sq[][MAX]` is a two-dimensional integer array whose row length is `MAX`. In C the second dimension of a two-dimensional array parameter must be written, because the compiler computes the address `sq[i][j]` as `i · MAX + j`. `MAX` is a constant defined in the program with `#define` and cannot be smaller than n.
- `i = 0`, `j = (n - 1) / 2` is the middle of the first row. For $n = 5$, $j = 2$. 1 is written there. The code assumes that all cells are zero; the caller must initialize the array to zero. Whether a cell is occupied is determined by checking whether it is nonzero.

- The loop places every number from $s = 2$ to n^2.
- `r` and `c` are the row and column of the upper-left diagonal. If `i - 1 < 0`, the row has gone off the top; `r` goes to the bottom row, `n - 1`. The same check is made for the column. The conditional operator `? :` writes this wraparound on one line.
- If `sq[r][c]` is nonzero, the diagonal cell is occupied. In that case one steps down from the current cell: `i = (i + 1) % n`. The modulo operation wraps an overflow at the bottom edge to the top row. `j` does not change.
- If the diagonal cell is empty, one moves there: `i = r; j = c;`. Then the number is written with `sq[i][j] = s`.

The code was compiled and run for $n = 3$, 5, and 7, and the sums of every row, column, and the two diagonals were checked with a piece of program. In all three squares the sums came out 15, 65, and 175 respectively. The output for the 5 × 5 square is the same as Figure 1.12.

1.8.4 Code trace: n = 3

Let us trace the call `magic(3, sq)`. Since the square is 3 × 3, the row and column indices take the values 0, 1, and 2. Table 1.15 gives where each number is placed and why.

`s`	Previous (i, j)	(r, c)	Diagonal	New (i, j)
1	—	—	middle of the first row	(0, 1)
2	(0, 1)	(2, 0)	empty; row overflowed → bottom row	(2, 0)
3	(2, 0)	(1, 2)	empty; column overflowed → rightmost column	(1, 2)
4	(1, 2)	(0, 1)	occupied (1 is there) → one down	(2, 2)
5	(2, 2)	(1, 1)	empty	(1, 1)
6	(1, 1)	(0, 0)	empty	(0, 0)
7	(0, 0)	(2, 2)	occupied (4 is there) → one down	(1, 0)
8	(1, 0)	(0, 2)	empty; column overflowed → rightmost column	(0, 2)
9	(0, 2)	(2, 1)	empty; row overflowed → bottom row	(2, 1)

Table 1.15: Trace of the call `magic(3, sq)`. (r, c) is the upper-left diagonal.

Two lines of the trace need care. At $s = 4$ the upper-left diagonal cell is (0, 1), and the 1 is there; so one steps down below the 3, to (2, 2). At $s = 7$ the current cell is (0, 0). The upper-left diagonal overflows both the row and the column and becomes (2, 2); the 4 is there. So one steps down below (0, 0), to (1, 0). When $s = 10 > 9$ the loop ends.

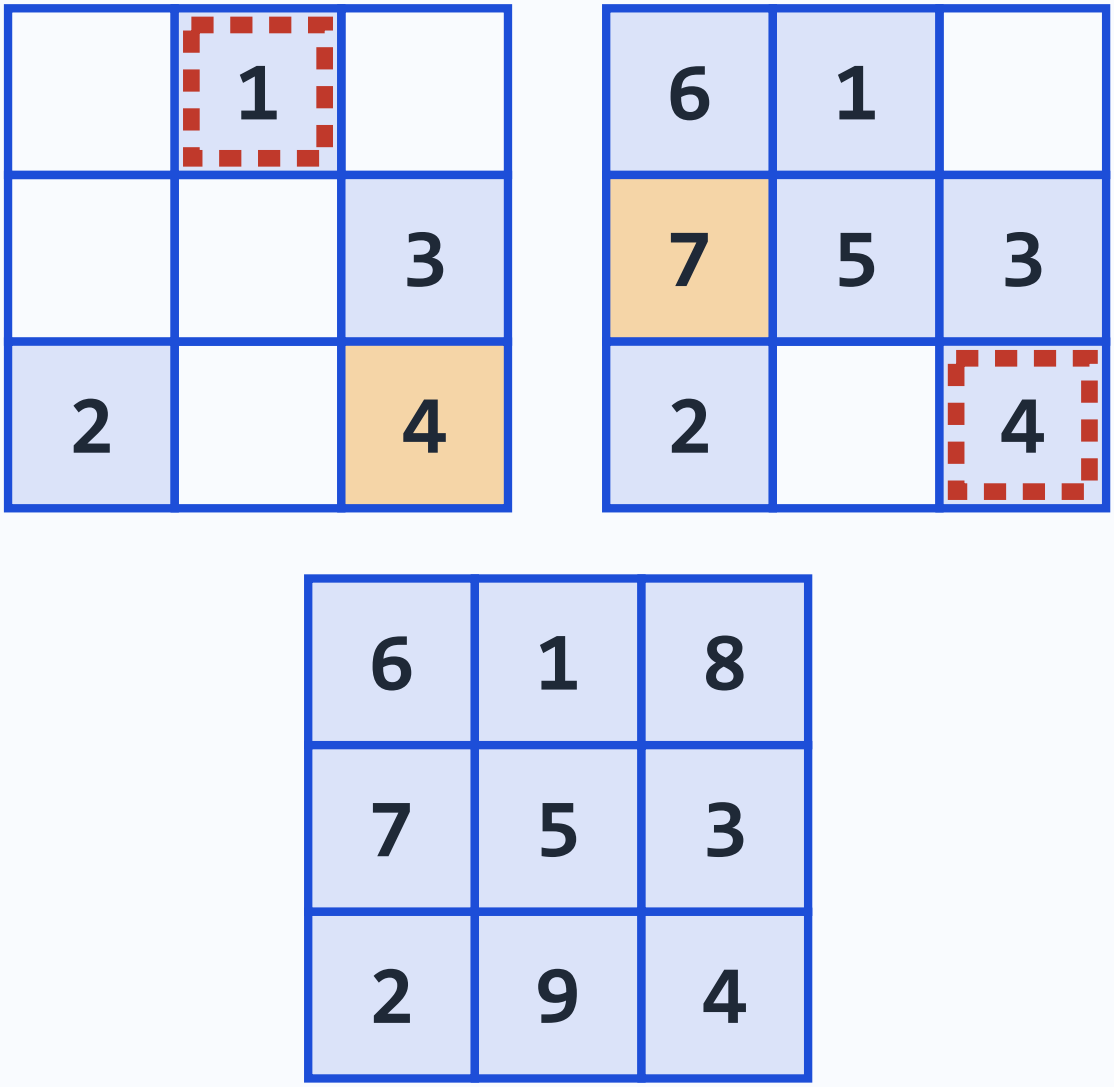


Figure 1.13: Three moments from the trace of `magic(3, sq)`. (a) *s = 4*: because the diagonal cell is occupied, 4 was written below 3. (b) *s = 7*: 7 was written below 6. (c) The finished square.

The resulting square is this: the first row is 6, 1, 8; the second row 7, 5, 3; the third row 2, 9, 4. The sum of every row, every column, and the two diagonals is $15 = 3 \cdot (9 + 1)/2$. For example the main diagonal is $6 + 5 + 4 = 15$ and the other diagonal is $8 + 5 + 2 = 15$.

1.9 Measuring performance

Performance analysis states the growth rate of an algorithm independently of the machine. But it does not say how many milliseconds it will take on a real machine with a real input. The way to find out is to run the program and measure its time. A measurement also includes the constants, cache effects, and compiler optimizations that the analysis ignores. Therefore, when two algorithms are in the same complexity class, only measurement tells which one is faster.

1.9.1 clock() and time()

The standard C library offers two functions for measuring time, in the header `<time.h>`. Table 1.16 compares how the two are used.

	1. Processor time: `clock()`	**2. Calendar time:** `time()`
Start	`start = clock();`	`start = time(NULL);`
Stop	`stop = clock();`	`stop = time(NULL);`
Type	`clock_t`	`time_t`
Result (seconds)	`(double)(stop - start) / CLOCKS_PER_SEC`	`difftime(stop, start)`

Table 1.16: Two ways of measuring time in C.

`clock()` returns the processor time spent since the start of the program, in **clock ticks**. The number of ticks per second is given by the constant `CLOCKS_PER_SEC`. The difference between two measurements is converted to seconds by dividing by this constant. The `(double)` cast must be applied before the division; otherwise the quotient of two integers drops the fraction and 0.4 seconds comes out as 0. The resolution of `clock()` is limited by the tick interval of the system: a job that finishes between two ticks seems to take 0 time.

`time()`, on the other hand, gives the wall clock, that is, calendar time, in seconds. `difftime` returns the difference between two times as a `double`. Its resolution is one second; it is suitable for jobs that last minutes and too coarse for short ones. `time()` also includes the time of other programs running on the computer at that moment.

The name `CLK_TCK` turns up in old books and old code. This name is nonstandard and a leftover from old compilers; most of today's compilers do not recognize it. In standard C, `CLOCKS_PER_SEC` is used.

There is one more rule when measuring: operations with peripherals, that is, writing to the screen, writing to disk, or sending to a printer, are very slow compared with the speed of the processor. If a `printf` is left inside the measured region, the measured time is largely the time of the `printf`; the time of the algorithm is overshadowed by it. `printf`s must be kept outside the measured region.

1.9.2 Timing selection sort

The program that measures the real time of selection sort on arrays of different sizes is this:

```c
#include <stdio.h>
#include <time.h>
#define MAX_SIZE 5001
#define ITERATIONS 20
void sort(int list[], int n);   /* selection sort */

int main(void) {
  int i, j, list[MAX_SIZE];
  int sizelist[] = { 0, 100, 300, 600, 900, 1200, 1500, 2000,
    3000, 3100, 3200, 3300, 3400, 3500, 3600, 3700, 3800,
    3900, 4000, 5000 };
  clock_t start, stop;  double ms;
  for (i = 0; i < ITERATIONS; i++) {
    for (j = 0; j < sizelist[i]; j++)
      list[j] = sizelist[i] - j;   /* descending array */
    start = clock();
    sort(list, sizelist[i]);
    stop = clock();
    ms = 1000.0 * (stop - start) / CLOCKS_PER_SEC;
    printf("%4d elements: %6.1f ms\n", sizelist[i], ms);
  }
  return 0;
}
```

- The array `sizelist` holds the 20 sizes to be measured: from 0 to 5000. The sizes are chosen densely between 3000 and 4000, so that the times of neighboring sizes can be compared.
- For each size the array is filled from largest to smallest with `list[j] = sizelist[i] - j`. Because the number of comparisons of selection sort is independent of the input, a descending array is not a special case; but it ensures that every measurement is made with the same kind of input.
- Between the two `clock()` calls there is only the call to `sort`. Filling the array and printing the result are outside the measured region.
- `ms = 1000.0 * (stop - start) / CLOCKS_PER_SEC` converts the time to milliseconds. `1000.0` is a `double` constant; so the whole expression is computed with `double` arithmetic. In old code this line appears in the form `(int)(1000*(stop-start)/CLK_TCK)`; integer division drops the fraction and `CLK_TCK` is nonstandard. That is why `double` arithmetic with `1000.0` is more correct.
- The definition of `sort` is not given in the program; the selection sort function and the `SWAP` macro must be added to the same file. `MAX_SIZE` is 5001; the largest measured size, 5000 elements, fits.

1.9.3 Results and discrepancies

A sample output of the program on an old machine is plotted in Figure 1.14. The dots show the measured times, and the dashed curve shows a curve $c \cdot n^2$. The general trend agrees with the analysis: time grows with n^2. But there are oddities in the details.

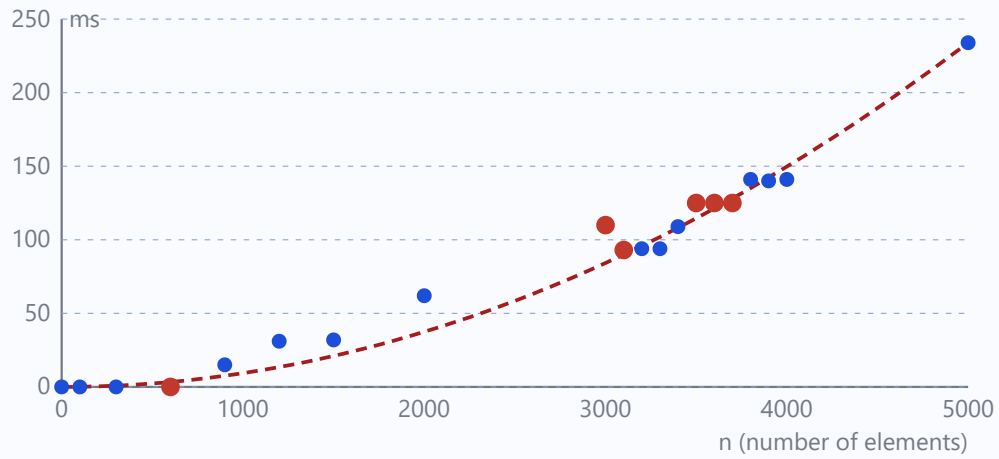


Figure 1.14: Single-run measurement results of selection sort. The dots are sample measurements, and the dashed curve shows the trend $c \cdot n^2$. The red dots are measurements that look inconsistent.

- Up to 600 elements, all measurements came out as 0 ms. The sorting did take time, but it took too little to be measured.
- 3000 elements took 110 ms, while 3100 elements took 93 ms. A larger input seems to have taken less time.
- 3500, 3600, and 3700 elements all took 125 ms; there is no difference among them.

The cause of these oddities is the resolution of `clock()`. The counter increases only at a clock tick. If the sorting starts and ends between two ticks, `stop - start = 0`. On a system with a tick interval of about 15.6 ms, all results are rounded to multiples of 15–16 ms: 93, 110, and 125 ms correspond to six, seven, and eight ticks respectively. The real time can deviate by as much as a tick. So the "reversed" difference between 3000 and 3100 elements is not real. In addition, the operating system can pause the program for a short time during the measurement and run another job; this also spoils single measurements.

The program was also run on a present-day machine, the one on which these notes were prepared. The results for selected sizes are given in Table 1.17. The code was compiled without optimization (`-O0`); on this machine the resolution of `clock()` is 1 ms. Times vary with the machine, the compiler, and the build options.

n	**300**	**600**	**1200**	**2000**	**3000**	**3500**	**3600**	**4000**	**5000**
ms	0	1	4	11	23	40	37	38	64

Table 1.17: Single-run measurement of selection sort on a present-day machine (ms; varies by machine).

The machine is many times faster than the old one, but the oddity persists: 3500 elements took 40 ms and 3600 elements took 37 ms. A resolution of 1 ms is still coarse for short measurements; single measurements are also open to interference from the operating system. The solution to this problem is to repeat every measurement many times and take the average.

Common mistake

Measuring a short job with a single `clock()` measurement. The result either comes out 0 or is rounded to multiples of a tick interval. A job that takes microseconds, such as a search, must be repeated thousands of times inside the measured region, and the total time divided by the number of repetitions.

1.9.4 Linear search with a sentinel

Let us see repeated measurement on linear search. The search function uses a small trick:

```
int seqsearch(int list[], int searchnum, int n) {
  int i;
  list[n] = searchnum;          /* sentinel */
  for (i = 0; list[i] != searchnum; i++)
    ;
  return (i < n) ? i : -1;
}
```

In the loop of an ordinary linear search, two tests are made in every iteration: `i < n` (has the end of the array been reached?) and `list[i] != searchnum` (has the sought value been found?). `seqsearch` removes the first test. It writes the sought value just past the end of the array, into `list[n]`. This cell is called a **sentinel**. Even if the sought value is not in the array, the loop stops at the sentinel at the latest; so no bounds check is needed. When the loop stops, if `i < n` then the value has really been found in the array and its index is returned. If `i == n`, the loop has stopped at the sentinel, and `-1` is returned to report that the value was not found.

The sentinel trick has a price: for n elements, $n + 1$ places must be reserved in the array. This is why `MAX_SIZE = 5001` was chosen in the measurement program. The sentinel also modifies the array; the old value in `list[n]` is lost. That is why the measurement program erases the sentinel after each size.

In the worst case, that is, when the sought value is not in the array, `seqsearch` makes $n + 1$ comparisons: n elements and the sentinel. The time is $O(n)$; the best case, when the sought value is in the first cell, is $\Theta(1)$. The sentinel does not change the number of comparisons, but it cuts the number of tests per iteration from two to one; the loop can become roughly twice as fast.

1.9.5 Code trace: seqsearch()

Let us trace the call `seqsearch(list, 5, 3)` on the array `list = {4, 8, 2}`. The array has three elements, but space is also reserved for `list[3]`. The value 5 is being sought.

1. `list[3] = 5`: the sentinel is put in place. The loop will now stop here at the latest (Figure 1.15a).

2. `i = 0`: `list[0] = 4 ≠ 5`, `i` becomes 1. Only one test is made; there is no bounds check.
3. `i = 1`: `list[1] = 8 ≠ 5`, `i` becomes 2.
4. `i = 2`: `list[2] = 2 ≠ 5`, `i` becomes 3.
5. `i = 3`: `list[3] = 5`; the loop stops. But the value found is the sentinel (Figure 1.15b).
6. `i = 3 < n` is false; the function returns `-1`: 5 is not in the array. In total *n + 1 = 4* comparisons have been made.

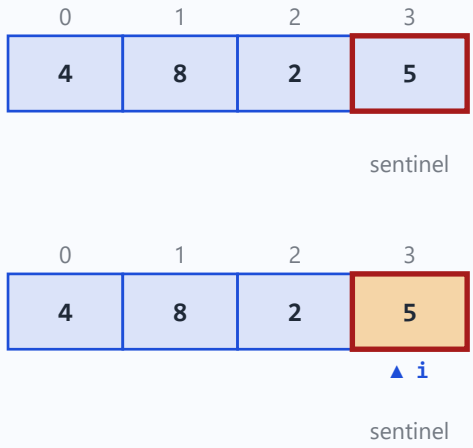


Figure 1.15: `seqsearch({4, 8, 2}, 5, 3)`. (a) The sentinel was written to `list[3]`. (b) The loop stopped at `i = 3`, on the sentinel; the result is `-1`.

If 8 is searched for in the same array, the loop stops at `i = 1`; since *1 < 3*, the function returns 1. These two calls were compiled and run, and the results were verified as `-1` and 1.

1.9.6 Repeated measurement

A search alone takes microseconds; its time cannot be found with a single measurement. The measurement program therefore repeats the search many times for each size. The measurement part of the program is this:

```
start = clock();
for (j = 0; j < runtimes[i]; j++)
  pos = seqsearch(list, -1, sizelist[i]);
stop = clock();
total = 1000.0 * (stop - start) / CLOCKS_PER_SEC;
avg = 1000.0 * total / runtimes[i];  /* microsec */
list[sizelist[i]] = sizelist[i];  /* erase sentinel */
```

- For every *n* the search is repeated `runtimes[i]` times. The repetition count is large for small sizes (500,000) and small for large sizes (5000). That way the total measurement time of each size is much longer than a few ticks, yet the measurement does not take minutes.

- `-1` is searched for. Since the array contains only numbers from 0 to $n - 1$, `-1` is never found; every search is in the worst case and makes exactly $n + 1$ comparisons.
- `total` is the total time in milliseconds. `avg = 1000.0 * total / runtimes[i]` gives the average time of a single search in microseconds: multiplying milliseconds by 1000 converts them to microseconds.
- The last line erases the sentinel: it writes the old value, that is n, back into `list[n]`. The array is assumed to have been filled with `list[j] = j`.

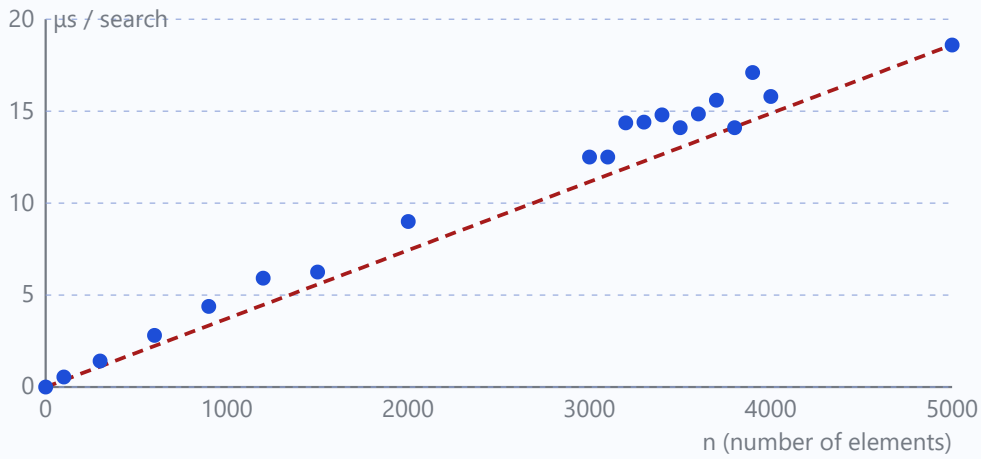


Figure 1.16: Average search times found by repeated measurement (µs per search). The dots are the measured averages, and the dashed line shows the linear trend.

Figure 1.16 shows the results. The averages are no longer 0 and grow linearly with n; this agrees with the expectation of $O(n)$. The fluctuation after 3000 is caused by the decrease in the repetition count: as `runtimes` gets smaller, the total time also gets shorter and the measurement becomes coarse again. The machine on which the measurement was made had 1000 ticks per second; that is, one tick is 1 ms.

The same measurement was also made while preparing these notes on a present-day machine. About 0.4 µs per search was measured at 100 elements, about 3.4 µs at 1000 elements, and about 18 µs at 5000 elements. The values vary by machine, but their ratios stay the same: when n grows tenfold, the time also grows about tenfold. The way to proceed when measuring short jobs, in brief, is this: the job is repeated many times, the total time is measured, and it is divided by the number of repetitions.

1.10 Recursive routines

The function `rsum` showed how a problem can be solved with a smaller copy of itself. The same approach makes it possible to translate directly into code many functions that are defined recursively in mathematics. Every recursive routine has two parts: a **base case** that gives a result without calling itself, and a **recursive step** that reduces the problem to a smaller one.

1.10.1 Factorial, Fibonacci, and the binomial coefficient

Let us take three classic functions:

- **Factorial.** $n! = n \cdot (n-1)!$ and $0! = 1! = 1$. The base case is $n \le 1$, and the step is $n \cdot fact(n-1)$.
- **The Fibonacci sequence.** $f_0 = 0$, $f_1 = 1$, and for $i > 1$, $f_i = f_{i-1} + f_{i-2}$. The base case is $n \le 1$ and the result is n itself; the step is $fib(n-1) + fib(n-2)$.
- **The binomial coefficient.** $C(n, m) = n! / (m! \cdot (n-m)!)$ is the number of ways of choosing m of n elements. The relation of Pascal's triangle is $C(n, m) = C(n-1, m) + C(n-1, m-1)$. The base case is $m = 0$ or $m = n$; in both the result is 1.

The definitions translate into C almost word for word:

```c
long fact(long n) {
  if (n <= 1) return 1;
  return n * fact(n - 1);
}

long fib(long n) {
  if (n <= 1) return n;   /* f0 = 0, f1 = 1 */
  return fib(n - 1) + fib(n - 2);
}

long binom(int n, int m) {
  if (m == 0 || m == n) return 1;
  return binom(n - 1, m) + binom(n - 1, m - 1);
}
```

When compiled and run, the code gave `fact(5) = 120`, `fact(10) = 3628800`, `fib(10) = 55`, `binom(5, 2) = 10`, and `binom(10, 3) = 120`. Let us trace the call `fact(3)`. `fact(3)` wants to compute 3 · `fact(2)`; `fact(2)` wants to compute 2 · `fact(1)`; `fact(1)` is the base case and returns 1. On the way back, `fact(2)` returns 2 · 1 = 2 and `fact(3)` returns 3 · 2 = 6. As in `rsum`, the stack first grows and then shrinks.

> **Common mistake**
>
> Writing `fib(n - 1) * fib(n - 2)` in the Fibonacci step. Since the base values are 0 and 1, the product always gives 0 or 1: $f_2 = 1 \cdot 0 = 0$, $f_3 = 0 \cdot 1 = 0$, and 0 from then on. By definition the step is an addition. A common mistake with the binomial coefficient is to write the function without parameters, as `long binom()`; n and m must be parameters, because every recursive call is made with different values.

1.10.2 Cost

`fact(n)` calls itself $n - 1$ times and does constant work in each call. Its time is $O(n)$; the stack depth is also n. The same job can be done with a loop in $O(n)$ time but with $O(1)$ memory.

`fib` and `binom`, in this form, take **exponential** time. The reason is clear: the same subproblems are solved again and again. `fib(5)` calls `fib(4)` and `fib(3)`; `fib(4)` in turn calls `fib(3)` again. `fib(3)` is computed twice and `fib(2)` three times. The total number of calls is $2f_{n+1} - 1$. In an experiment with a counter, `fib(10)` made 177 calls, `fib(20)` made 21,891, and `fib(30)` made 2,692,537. With every increase of 10, the number of calls grows by a factor of about 123. The stack depth, in contrast, is only n, because at most n frames are on the stack at the same time. Fibonacci numbers can be computed in $O(n)$ time with a loop that keeps the last two values. Reducing exponential time to linear or polynomial time by storing the results of repeated subproblems is one of the important ideas of algorithm design.

1.11 Complexity summary

The time and extra memory requirements of the routines examined in these notes are collected in Table 1.18.

Routine	Time	Extra memory	Remark
`sort` (selection sort)	$\Theta(n^2)$	$O(1)$	$n(n-1)/2$ comparisons, at most $n - 1$ swaps
`sum`	$\Theta(n)$	$O(1)$	$2n + 3$ steps
`rsum`	$\Theta(n)$	$O(n)$	$2n + 2$ steps, $n + 1$ frames
`add` (matrix addition)	$\Theta(rows \cdot cols)$	$O(1)$	$\Theta(n^2)$ for a square matrix
`seqsearch`	$O(n)$	$O(1)$	best $\Theta(1)$, worst $n + 1$ comparisons
`magic` (Coxeter)	$\Theta(n^2)$	$O(1)$	brute force: $(n^2)!$ arrangements
dynamic array insertion	$O(1)$ amortized	$O(n)$	a single insertion is $O(n)$ in the worst case
`fib` (recursive)	exponential	$O(n)$	stack depth n

Table 1.18: Complexities of the routines examined in these notes.

The "extra memory" column in the table does not count the input itself. `sort`, `sum`, and `add` use only a few local variables. `rsum` and `fib` take space proportional to n on the call stack. The $O(n)$ memory of the dynamic array is the block that holds the elements itself: the capacity is at most twice the number of elements, and during reallocation the old and new blocks briefly coexist.

1.12 Common mistakes

1. **Assuming "it compiled and ran, so it is correct".** A program may give the right result on one example yet work wrongly on empty input, a single element, or repeated values. The test data must cover all scenarios. The faulty version of `sort` likewise gives correct results but is not the intended algorithm.
2. **Confusing the step count with constants.** $2n + 3$ and $100n$ are in the same class: both are $O(n)$. Asymptotic notation deliberately drops constant factors and small terms.
3. **Taking a loose upper bound for the best bound.** $3n + 2 = O(n^2)$ is true but tells nothing. The smallest possible $g(n)$ is written; if a tight bound is known, Θ is used.
4. **Measuring a short job with a single `clock()` run.** The result is 0 or a multiple of the tick interval. The job must be repeated many times and the average taken.
5. **Leaving a `printf` in the measured region.** The input/output time overshadows the time of the algorithm.
6. **Giving a macro an argument with a side effect.** `SWAP(a[i++], a[j], t)` increments `i` twice and destroys a value.
7. **Not checking the result of `malloc` and not giving back allocated memory.** The first leads to null pointer access, the second to a memory leak.

1.13 Summary

- **Process.** Software goes through the phases of requirements, analysis, design, coding, and verification. In design, data objects are defined as ADTs and operations as algorithms. Postponing the implementation makes it possible to choose better.
- **Algorithm.** A finite sequence of instructions satisfying the criteria of input, output, definiteness, finiteness, and effectiveness. Programs that do not terminate are computational procedures.
- **Abstraction.** An ADT separates the specification from the implementation. In C the tools for this are the header file and the opaque type: the user holds only a `Fraction *`, and the fields are hidden in the `.c` file. The contract and the ownership rule of the interface must be written down explicitly.
- **Selection sort.** In each round the smallest of the remaining part is moved to the front. It makes $n(n - 1)/2$ comparisons on every input and is $\Theta(n^2)$; the number of swaps is at most $n - 1$. The swap is done by passing the addresses of the variables, with a function or with a macro.
- **Counting.** Memory is $S(P) = c + S_P(I)$. Time is counted in program steps: `sum` takes $2n + 3$, `rsum` $2n + 2$, and `add` $2 \cdot rows \cdot cols + 2 \cdot rows + 1$ steps. Recursion opens a frame on every call and uses memory.

- **Classification.** O is the upper, Ω the lower, and Θ the tight bound. The worst case is usually given, because it is a guarantee for every input. Amortized analysis looks at the total cost of a sequence of operations; in a dynamic array that grows by doubling, insertion is amortized $O(1)$.
- **Growth.** Small n misleads. 2^n and $n!$ become hopeless in no time. In the magic square, brute force takes millions of centuries and Coxeter's rule takes microseconds: the algorithm comes before the hardware.
- **Measurement.** Time is measured with `clock()`; short jobs are repeated and averaged; input/output is kept outside the measured region. A sentinel removes the bounds test in every iteration of linear search.

1.14 Key terms

system life cycle bottom-up top-down algorithm abstract data type (ADT) implementation
header file incomplete type opaque type struct typedef arrow operator (->) malloc free
null pointer (NULL) memory leak dangling pointer sorting selection sort swap macro
preprocessor dereference (*) performance analysis space complexity time complexity
program step recursion base case call stack stack frame stack overflow
best / average / worst case asymptotic notation Big O Big Omega Big Theta exponential
amortized analysis dynamic array capacity realloc magic square brute force
linear (sequential) search sentinel

1.15 Test Yourself

1. In the fragment `for (i = 0; i < n; i++) for (j = i; j < n; j++) x++;`, how many times is `x++` executed? What is the value for $n = 4$, and which class is the time of the fragment in?

   Answer

   In round `i` the inner loop runs $n - i$ times. The total is $n + (n - 1) + \ldots + 1 = n(n + 1)/2$. For $n = 4$, $4 \cdot 5 / 2 = 10$. The fragment is $\Theta(n^2)$.

2. In the claim $5n^2 + 20n + 50 = O(n^2)$, if $c = 6$ is chosen, what is the smallest n_0?

   Answer

   The inequality $5n^2 + 20n + 50 \le 6n^2$ is equivalent to $20n + 50 \le n^2$. At $n = 22$, $490 > 484$; at $n = 23$, $510 \le 529$, and it holds from then on. The smallest n_0 is 23. If $c = 75$ is chosen, $n_0 = 1$ suffices.

3. How many program steps does the following function, which finds the largest element of an array, take in the worst and in the best case? `int find_max(int a[], int n) { int m = a[0]; int i; for (i = 1; i < n; i++) if (a[i] > m) m = a[i]; return m; }`

   Answer

   The assignment is 1 step, the `for` test n times, the `if` test $n - 1$ times, and the `return` 1 step. The assignment `m = a[i]` is executed $n - 1$ times in the worst case (when the array is in ascending order) and not at all in the best case (when the largest element is first). The worst case is $3n$ and the best case $2n + 1$ steps; both are $\Theta(n)$. The version with a counter gives 18 and 13 for $n = 6$.

4. How many comparisons and how many real swaps (`i ≠ min`) does selection sort make on the array {5, 4, 3, 2, 1}?

   Answer

   The number of comparisons is independent of the input: $5 \cdot 4 / 2 = 10$. Swaps: at `i = 0` 5 and 1 are exchanged, at `i = 1` 4 and 2; the array becomes {1, 2, 3, 4, 5}. At `i = 2` and `i = 3`, `min = i`. In total 2 real swaps are made.

5. How many comparisons does `seqsearch` with a sentinel make on an array of five elements when searching for a value that is not in the array? How many when the sought value is at index 2?

   Answer

   If the value is absent, $n + 1 = 6$ comparisons are made, with the five elements and the sentinel. If the value is at index 2, 3 comparisons are made, with `list[0]`, `list[1]`, and `list[2]`; the loop stops at `i = 2` and since $2 < 5$, 2 is returned.

6. 20 elements are appended to an empty dynamic array of capacity 4, with the capacity doubling each time it fills. How many elements are copied in total? What is the total cost including the writes?

   Answer

   At the 5th insertion 4 elements are copied, at the 9th 8, and at the 17th 16: 28 in total. The total cost with the writes is $28 + 20 = 48$; this comes to 2.4 operations per insertion and stays below the bound $3n = 60$.

7. How many times in total does the recursive call `fib(5)` call `fib`, including the first call? How

many `fib` frames are on the stack at the deepest moment?

Answer

The number of calls is $2f_6 - 1 = 2 \cdot 8 - 1 = 15$. It can also be counted by hand: from the relation $T(n) = T(n-1) + T(n-2) + 1$, $T(0) = T(1) = 1$, one gets $T(2) = 3$, $T(3) = 5$, $T(4) = 9$, $T(5) = 15$. Since the deepest chain is `fib(5)` → `fib(4)` → `fib(3)` → `fib(2)` → `fib(1)`, there are at most 5 frames on the stack.

8. A $\Theta(n^2)$ algorithm takes 1 second on 1000 elements. About how long does it take on 4000 elements? How many elements can be processed in 1 second on a machine ten times faster?

Answer

When n quadruples, the time grows by a factor of $4^2 = 16$: about 16 seconds. On a machine ten times faster, ten times as many steps are taken in 1 second; if n^2 grows tenfold, n grows by a factor of $\sqrt{10} \approx 3.16$. About 3160 elements are processed. For a quadratic algorithm, making the machine ten times faster enlarges the input only threefold.

9. In a 5 × 5 magic square built with Coxeter's rule, into which cell is the number 2 written? Why?

Answer

1 is written in the middle of the first row, at (0, 2). The upper-left diagonal cell is (−1, 1); since the row overflows the top, it wraps to the bottom row and becomes (4, 1). The cell is empty; 2 is written there. In Figure 1.12 the 2 can be seen in the second cell of the bottom row.

10. What happens if `Fraction f; f.num = 3;` is written in `main.c`? Why?

Answer

A compile error results. `main.c` sees only `fraction.h`, and there `struct fraction` is an incomplete type: the compiler does not know its size or fields. Therefore a variable of this type cannot be defined and its fields cannot be accessed. The user can work with fractions only through a `Fraction *` pointer and the functions in the interface; that is the purpose of the opaque type.

1.16 Further reading

- E. Horowitz, S. Sahni, and S. Anderson-Freed, *Fundamentals of Data Structures in C* (2nd ed.), Chapter 1: the basis of these notes; it presents the system life cycle, the ADT, step-count tables, asymptotic notation, the magic square, and performance measurement with more exercises.

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms* (3rd ed.), Chapters 2–3 and 17: defines algorithm analysis and the notations O, Ω, and Θ precisely and with proofs, and treats amortized analysis in detail with the dynamic table example.
- R. Sedgewick, *Algorithms in C*: presents the principles of algorithm analysis with C code and many measurements; shows how theoretical predictions are compared with experiments.
- B. W. Kernighan and D. M. Ritchie, *The C Programming Language* (2nd ed.): the reference for pointers, structures, the preprocessor, and the standard library (`<stdlib.h>` , `<time.h>`).

1.17 References

1. E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2nd ed., Chapter 1.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 3rd ed.: Chapters 2–3 (analysis, asymptotic notation), Chapter 17 (amortized analysis).
3. B. W. Kernighan, D. M. Ritchie, *The C Programming Language*, 2nd ed.
4. R. Sedgewick, *Algorithms in C*.

Arrays and Structures

The array is the oldest and most widely used data structure in programming. Its elements are reached by index in constant time. This speed comes from the array sitting contiguously in memory and from the address of every element being computed with a simple formula. The chapter first defines the array as an abstract data type and then descends to its implementation in C. The relationship between arrays and pointers follows, then arrays that grow with `malloc`, `realloc`, and `free`, and the amortized cost of doubling. How multidimensional arrays are laid out in one-dimensional memory, and how the address of an element is computed, is treated in detail. The same idea is used to hold lower triangular and band matrices by storing only the necessary elements. The last sections cover the `struct` and `union` types, which bring together fields of different types, C strings, and the analysis of matrix multiplication.

LEARNING OUTCOMES

See the array abstractly and concretely
Define the array ADT and explain the contiguous layout of a C array in memory and the relationship between an array name and a pointer.

Compute addresses
Find the address of an element of two-, three-, and *n*-dimensional arrays in row-major and column-major layout; reduce the number of multiplications with Horner's method.

Save memory
Store triangular and band matrices in a one-dimensional array and write functions that reach every element in *O(1)* time.

Dynamic memory
Build growing arrays and two-dimensional arrays with `malloc`, `realloc`, and `free`; justify the amortized cost of doubling.

Record types
Use `struct`, `union`, and strings terminated by `'\0'` correctly and safely; explain the effect of alignment on structure size.

Matrix operations
Derive the step counts of matrix addition and multiplication and show that multiplication is $O(n^3)$.

2.1 The array abstract data type

A data structure can be thought of at two levels. At the abstract level one speaks only of what the structure does, that is, which values it holds and which operations it offers. At the concrete level one deals with the layout in memory and the code with which those operations are carried out. The first level is called the **abstract data type** (ADT), the second the **implementation**. The array is a structure in which the two levels are often confused: say "array" and one immediately thinks of cells sitting side by side in memory. Yet sitting side by side is not the definition of the array but its most common implementation.

2.1.1 An array is a mapping

Viewed abstractly, an array is a relationship established between indices and values. Each valid index corresponds to exactly one value. The grades of the students in a class by their sequence number, the temperatures by the days of a year, or the pieces by the squares of a chessboard are relationships of this kind. The index set may be a range of numbers or, as on a chessboard, may consist of pairs of numbers.

Definition 2.1

An **array** is a set of ⟨index, value⟩ pairs. For every defined index, the set contains exactly one pair that starts with that index. This relationship is called a **mapping**: an array is a mapping from the index set to the value set.

An ADT is defined not only by its values but also by its operations. The textbook by Horowitz, Sahni, and Anderson-Freed gives three operations for the array ADT. `Create` builds an empty array, `Retrieve` reads the value at an index, and `Store` writes a value at an index. The meaning of these operations is shown in Table 2.1.

Operation	Meaning
`Create(j, list)`	Returns an empty array of dimension *j* whose index set is `list`.
`Retrieve(A, i)`	Returns the value *A[i]* if *i* is a valid index, and an error otherwise.
`Store(A, i, x)`	If *i* is valid, pairs *i* with *x* in *A* and returns the new array; otherwise returns an error.

Table 2.1: The operations of the array ADT.

Two details of the definition are important. First, `Retrieve` and `Store` give an error at an invalid index. At the ADT level, index checking is part of the operation. Second, `Store` returns "the new array". At the abstract level a write does not destroy the old array but produces a new mapping. In a language such as C, however, a write changes the contents of the existing cell. These two differences show why the implementation must be thought of separately from the ADT. A C array does not check the index at all and changes the place it writes to in place. These two properties give speed, and in return give up safety.

2.1.2 The source and the price of speed

The most common implementation of an array is to place the elements in consecutive cells of memory. Since every element has the same type, each occupies the same number of bytes. If this number is called *s* and the address of the first element *α*, the address of the element with index *i* is computed directly:

> $$\text{address}(A[i]) = \alpha + i \cdot s$$

This computation consists of one multiplication and one addition. Whatever the length of the array, the number of operations does not change. That is why access by index takes *O(1)* time. Reaching the 999th element of an array of a thousand elements takes as little time as reaching the first. In structures such as the linked list, one must start at the beginning and move element by element to reach the same element. This is the distinguishing strength of the array.

Contiguous placement has two costs. First, the size of the array is fixed in advance. The block allocated in memory does not grow by itself afterwards; if a bigger array is needed, a new block is allocated and the old elements are moved there. Second, insertion and deletion in the middle are expensive. To insert a new element at the front of an array of *n* elements, all the elements must be shifted one cell to the right. In deletion, too, the following elements are shifted left to close the gap. Since the number of shifted elements is *n* in the worst case, insertion and deletion in the middle take *O(n)* time. Appending at the end of the array requires no shifting; the cost of that operation is examined separately under dynamic arrays.

2.2 One-dimensional arrays in C

C implements the array ADT directly within the language. When an array is declared, the compiler allocates a contiguous block of memory and remembers the address of the first element. This section covers the declaration, the address computation, the `sizeof` operator, and the close relationship between the array name and the pointer. To follow the topic, it is enough to know two pointer operators: `&x` gives the address of the object `x`; `*p` reaches the object that the pointer `p` points to.

2.2.1 Declaration, address, and sizeof

The following declaration defines two arrays. `list` holds five integers, and `plist` holds five pointers to integers.

```
int list[5], *plist[5];
/* list : 5 integers          */
/* plist: 5 pointers to int   */

list[i]  /* address: α + i * sizeof(int) */
sizeof list   /* 5 * sizeof(int) bytes */
```

The compiler allocates a contiguous block of five `int`s for `list` and keeps the address of `list[0]`, that is, α. When the program writes `list[i]`, the address of the element is computed at run time as $\alpha + i \cdot$ `sizeof(int)`. `sizeof` is an operator that gives the size of a type or object in bytes. Its value is known at compile time. For a local array, `sizeof list` gives the size of the whole array: if `int` is 4 bytes, $5 \cdot 4 = 20$ bytes. `plist` holds five pointers, and on a 64-bit system each pointer is 8 bytes, so `sizeof plist` is 40. These values were measured in the driver program written for these notes; the sizes of `int` and of pointers may vary with the compiler and the system.



Figure 2.1: An array of four integers in memory. Assuming `sizeof(int) = 4`, the elements sit at the addresses α, $\alpha + 4$, $\alpha + 8$, and $\alpha + 12$.

Figure 2.1 shows this layout. Each box is an element; the index is written above the box and the address below it. The addresses grow by four bytes, because every `int` occupies four bytes. Indices start from 0: the valid indices of an array of n elements are $0, 1, \ldots, n - 1$. The reason for starting at zero is the address formula. The address of the first element becomes $\alpha + 0 \cdot s = \alpha$, and the formula needs no extra correction term.

C does not check whether an index is valid. While `Retrieve` and `Store` in the ADT give an error at an invalid index, C computes the address $\alpha + 5 \cdot 4$ for the expression `list[5]` as well and accesses it. No part of the array is at this address; it may be another variable, a return address, or a region that was never allocated. The C standard regards this case as **undefined behavior**. The program may crash, produce a wrong result, or seem to work correctly. The last is the most dangerous, because the error shows up only on another machine or with another input.

Common mistake

Writing the loop in an array of n elements as `for (i = 0; i <= n; i++)`. In the last round `a[n]` is accessed and one cell beyond the array is read or overwritten. The compiler does not catch this error. The correct condition is `i < n`; the rule "n elements, last index $n - 1$" must be kept in mind in every loop.

2.2.2 Array name and pointer

A **pointer** is a variable that holds the memory address of an object. In C, an array name turns into the address of the first element in most expressions. For example, when `list` is written, the compiler treats it as `&list[0]`, that is, as an address of type `int *`. This rule brings arrays and pointers very close together. Table 2.2 lists equivalent expressions.

Expression	Equivalent
`list`	`&list[0]`
`list + i`	`&list[i]` ($i \cdot$ `sizeof(int)` bytes forward)
`*(list + i)`	`list[i]`
`p = list;` then `p[i]`	the same cell as `list[i]`

Table 2.2: Equivalents of array and pointer expressions (`int list[5]; int *p;`).

The second row of the table contains the basic rule of **pointer arithmetic**. When an integer is added to a pointer, the unit is not a byte but one element of the type pointed to. If `p` is an `int *`, `p + 1` points to the next `int`; the address grows not by 1 but by 4 bytes. For `double *` the increase is 8 bytes, for `char *` 1 byte. The compiler does the multiplication itself. In this way the notation `*(list + i)` becomes the direct counterpart in C of the address formula $\alpha + i \cdot s$. In fact, the notation `list[i]` is defined in the standard as `*(list + i)`; the square brackets are a short way of writing this addition and dereference.

An array name and a pointer variable are still not the same thing. Two differences matter:

- **An array name cannot be assigned to.** `list = p;` does not compile. An array name is not a variable but a constant address known to the compiler. `p = list;` is valid, because `p` is a variable and can hold another address.
- `sizeof` **gives a different result.** For a local array, `sizeof list` gives the size of the whole array (20 bytes). For a pointer, `sizeof p` is only the size of the pointer itself (8 bytes on a 64-bit system).

The second difference often leads to mistakes in function parameters. In C, when an array is passed to a function, the array is not copied; only the address of its first element is passed. So the declarations `void f(int a[])` and `void f(int *a)` are exactly the same. Even if the parameter is written as `int a[10]`, `a` is a pointer inside the function and `sizeof a` gives 8. In the driver written for these notes,

the compiler issued the warning `-Wsizeof-array-argument` for this usage. The length of the array is sent to the function as a separate parameter. This is why functions of the standard library such as `qsort`, `memcpy`, and `fwrite` all take a length parameter.

Common mistake

Computing the array length inside a function with `sizeof a / sizeof a[0]`. This pattern works correctly only where the array is declared. Since `a`, arriving as a parameter, is a pointer, the result is *8 / 4 = 2* on a 64-bit system and has nothing to do with the real length of the array.

2.2.3 Code trace: print_array()

The following program passes an array of four elements to the function `print_array`. The function prints the address and value of each element using pointer arithmetic.

```c
void print_array(int *p, int n) {
  int i;
  for (i = 0; i < n; i++)
    printf("%p %d\n", (void *)(p + i), *(p + i));
}

int main(void) {
  int arr[] = {10, 20, 30, 40};
  print_array(arr, 4);
  return 0;
}
```

The program works as follows, line by line:

- `main` allocates `arr` on the stack as four consecutive `int`s. The size is derived from the number of initial values. Let us call the address of the first element α.
- In the call `print_array(arr, 4)` the array is not copied. The name `arr` turns into `&arr[0]`, and this address, that is α, is copied into the parameter `p`. The length arrives as a separate parameter (`n = 4`).
- In each round of the loop, `p + i` computes the address $\alpha + i \cdot 4$. `*(p + i)` reads the value at this address; it is the same as writing `p[i]`.
- The `%p` format expects a `void *`. That is why the address is converted with `(void *)`. If the conversion is not written, the behavior is undefined according to the standard.

`i`	`i < n`	`p + i`	`*(p + i)`	**Line printed**
0	0 < 4, true	$\alpha + 0$	10	α 10
1	1 < 4, true	$\alpha + 4$	20	$\alpha + 4$ 20

`i`	`i < n`	`p + i`	`*(p + i)`	**Line printed**
2	2 < 4, true	*α + 8*	30	*α + 8* 30
3	3 < 4, true	*α + 12*	40	*α + 12* 40
4	4 < 4, false	—	—	the loop ends, `print_array` returns

Table 2.3: Trace of the call `print_array(arr, 4)` (`sizeof(int) = 4`).

In the real output a hexadecimal address appears in place of *α*. In one run made for these notes the addresses came out `…F910` , `…F914` , `…F918` , and `…F91C` . The last two hexadecimal digits grow by four, as 10, 14, 18, 1C. The address itself may change on every run; what does not change is the 4-byte difference between two consecutive elements. The function visits each element of the array once and does constant work in each round. Its time is *Θ(n)* and its extra memory *O(1)*.

2.3 Dynamic arrays

In a declaration such as `int list[5];` the size is known at compile time. In many programs, however, the size is learned only at run time: the number of students the user enters, the number of lines in a file, or the number of packets arriving from a network connection. In this case memory is requested at run time. This is called **dynamic memory allocation**. In C it is done with the functions `malloc` , `calloc` , `realloc` , and `free` in the header `<stdlib.h>` .

2.3.1 An array with malloc

The following fragment first reads the size of the array, then allocates a block of that many `int` s, uses the block like an array, and finally gives it back.

```
int n, i, *a;
scanf("%d", &n);
a = malloc(n * sizeof *a);   /* n ints */
if (a == NULL) {
  fprintf(stderr, "Out of memory\n");
  exit(EXIT_FAILURE);
}
for (i = 0; i < n; i++)
  a[i] = i * i;              /* like an array */
free(a);                     /* give it back */
a = NULL;          /* no dangling pointer */
```

The code works as follows, line by line:

- `malloc(n * sizeof *a)` asks for the number of bytes sufficient for *n* integers and returns the starting address of the block. The return type is `void *` ; C assigns this address to `a` , of type `int *` , without a cast.

- If memory is insufficient, `malloc` returns `NULL`. A **null pointer** points to no object. If `a[i]` is written without checking this case, the program crashes. If the check fails, a message is written and the program terminates with `exit(EXIT_FAILURE)`.
- The loop fills the block with `a[i] = i * i`. Although `a` is a pointer, the notation `a[i]` works as with an array, because `a[i]` already means `*(a + i)`.
- `free(a)` gives the block back to the system. Then `a = NULL` is written. After `free`, `a` still holds the old address, but the memory at that address no longer belongs to the program. Such a pointer is called a **dangling pointer**. Assigning `NULL` turns a possible misuse into an immediately noticeable error instead of an unclear corruption.

A local array and a `malloc` block have different lifetimes. An array declared inside a function sits in the **stack** frame of the call and disappears when the function returns. A block obtained with `malloc`, on the other hand, sits in **heap memory** and lives until it is `free`d. A function can therefore safely return the address of a block it allocated inside, but cannot return the address of a local array. The address of a local array becomes a dangling pointer the moment the function returns.

The pattern `sizeof *a` in the size expression is a deliberate choice. `sizeof *a` gives the size of the type that `a` points to. Even if the type of `a` is later changed to `double *`, the expression computes the correct size by itself. Writing `sizeof(int)` is correct too, but one may forget to update it when the type changes. A more dangerous mistake is to confuse `sizeof(int *)` with `sizeof(int)`: on a 64-bit system the first is 8 and the second 4 bytes. If the wrong one is smaller, the end of the block is overrun; if it is larger, memory is wasted.

The library has two more functions. `calloc(n, s)` reserves room for *n* elements of *s* bytes each and zeroes all bytes of the block. `malloc` does not zero the block; the initial contents are indeterminate. `realloc(p, new_size)` enlarges or shrinks a previously allocated block. If the block cannot grow in place, `realloc` allocates a new block, copies the old contents there, and frees the old block. If `p` is `NULL`, `realloc` behaves exactly like `malloc`. This last property is useful in growing arrays that start empty.

> **Common mistake**
>
> Not checking the return of `malloc`, or continuing to use the pointer after `free`. In the first case, when memory is insufficient, the program tries to write to address `NULL`. In the second, the block that was given back may have been allocated again for another request, and a write lands on top of other data. Checking for `NULL` after every `malloc`, and assigning `NULL` to the pointer after every `free`, should become a habit.

2.3.2 A growing array: append()

If it is not known at all how many elements will come, the array must be able to grow. A **dynamic array** is an array that is enlarged with `realloc` when it fills up. Defining a dynamic array takes three pieces of information: the address of the block in which the elements sit, how many elements are

currently in the array, and the maximum number of elements the block can hold. The last value is called the **capacity**. In the following code these three pieces of information sit in the fields `data`, `size`, and `cap` of the structure `DynArray`.

```
typedef struct { int *data; int size, cap; } DynArray;

int append(DynArray *d, int x) {
  if (d->size == d->cap) {              /* full */
    int ncap = d->cap ? 2 * d->cap : 1;
    int *t = realloc(d->data, ncap * sizeof *t);
    if (t == NULL) return 0;    /* old block intact */
    d->data = t;
    d->cap = ncap;
  }
  d->data[d->size++] = x;
  return 1;
}
```

The code works as follows, line by line:

- `d` is a `DynArray *`. A pointer is taken so that the structure itself is not copied and the function can change `size` and `cap` permanently. The notation `d->size` is short for `(*d).size`.
- If `d->size == d->cap`, the array is full and must first be enlarged. The new capacity is twice the old one. If the capacity is 0, its double would also be 0, so `ncap` is taken as 1. The conditional operator `?:` separates these two cases in one line.
- The result of `realloc` is first taken into a temporary variable `t`. On failure `realloc` returns `NULL` but does not touch the old block. Had the result been written directly into `d->data`, the address of the old block would have been lost: the elements would be neither reachable nor freeable. This is a **memory leak**. In the code, in case of failure 0 is returned and the array stays intact in its old state.
- If the enlargement succeeds, `d->data` and `d->cap` are updated.
- The line `d->data[d->size++] = x;` writes the element into the first empty cell. `size++` is the postfix form: in the expression the old value is used first, and then the value is incremented by one. So `x` is written to `data[size]` and then `size` grows by one.

Example 2.1

Let 7, 3, 9, 4, and 6 be appended in turn to an empty dynamic array (`{NULL, 0, 0}`). At each append it is checked whether the array is full; if it is, the capacity is doubled and the old elements are copied to the new block. Table 2.4 gives all the steps.

x	size == cap	ncap	Copies (total)	cap	size	data
7	0 == 0, full	1	0 (0)	1	1	[7]

x	size == cap	ncap	Copies (total)	cap	size	data
3	1 == 1, full	2	1 (1)	2	2	[7, 3]
9	2 == 2, full	4	2 (3)	4	3	[7, 3, 9, _]
4	3 == 4, room left	—	0 (3)	4	4	[7, 3, 9, 4]
6	4 == 4, full	8	4 (7)	8	5	[7, 3, 9, 4, 6, _, _, _]

Table 2.4: Trace of `append` while 7, 3, 9, 4, 6 are appended to an empty array. "Copies" is the number of elements moved to the new block when growing.

At the first append `data` is `NULL` and `realloc(NULL, …)` behaves like a `malloc`; there are no elements to copy. At the later enlargements 1, 2, and 4 elements are copied in turn. For five appends only 7 copies are made in total. At the fourth append (4) there is room in the array and the element is written directly. In the driver program written for these notes, the number of `realloc` calls and the total of copies came out exactly the same as in this table. Here "copies" is the number of elements `realloc` moves in the worst case. If the block can grow in place, `realloc` does not copy at all; the analysis therefore gives an upper bound.

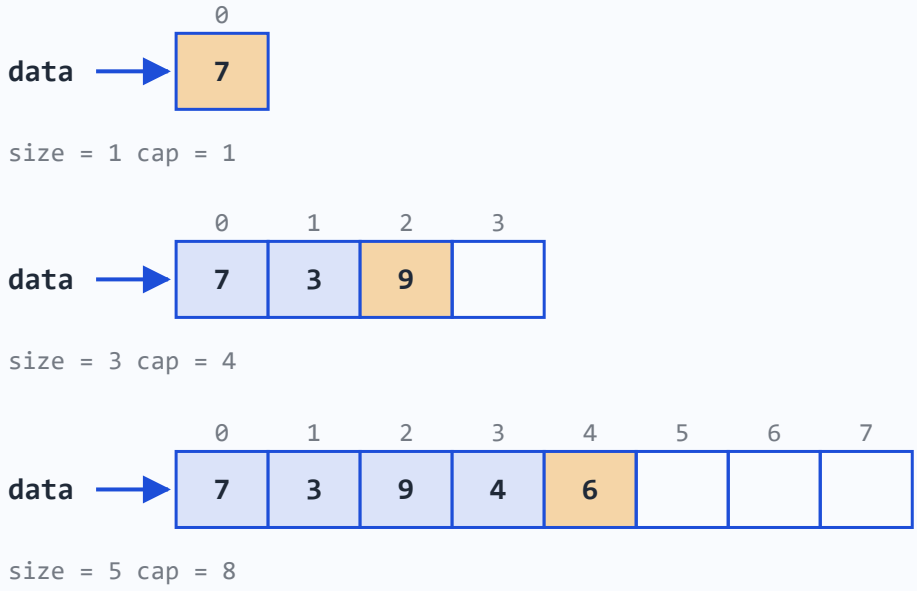


Figure 2.2: Three moments of the dynamic array. (a) First append: capacity 1, element 7. (b) Third append: the capacity has doubled to 4, 9 was written; one cell is empty. (c) Fifth append: capacity 8, five elements full, three cells empty.

2.3.3 Amortized cost

The cost of a single `append` call varies greatly. If there is room in the array, the call consists of one write and takes *O(1)* time. If the array is full, `realloc` may copy all the elements; that call takes *O(n)* time. Looking only at the worst case gives a pessimistic result such as $O(n^2)$ for n appends. Yet the

expensive calls are rare: because the capacity doubles each time, enlargement is needed only after 1, 2, 4, 8, … elements. Finding the cost per operation by dividing the total cost of a long sequence of operations by the number of operations is called **amortized analysis**.

Theorem 2.1 (Cost of doubling)

If $n \geq 1$ elements are appended to an empty dynamic array with the capacity doubling, the total number of copies is less than $2n$. Counting the writes as well, the total work is less than $3n$. Hence the amortized cost per append is $O(1)$.

Proof. Enlargements happen when 1, 2, 4, …, 2^k elements fill the array, where 2^k is the largest power of two less than n. At each enlargement the elements then in the array are copied. The total number of copies is a geometric series: $1 + 2 + 4 + \ldots + 2^k = 2^{k+1} - 1$. Since $2^k < n$, this total is less than $2n - 1$. Each element is also written once; that makes n units of work. The total is less than $3n$, and divided by n a constant per append remains. ■

Increasing the capacity by one cell each time, instead of doubling it, looks more economical at first glance. But in that case an enlargement is needed at every append. At the ith append $i - 1$ elements are copied, and the total number of copies is $0 + 1 + \ldots + (n - 1) = n(n - 1)/2$. This is $O(n^2)$. Growing by a fixed c does not fix the situation either: the number of enlargements is n/c, and the total number of copies is still proportional to n^2. What matters is that the increase is proportional to the current capacity, not fixed. Table 2.5 compares the two strategies for a few values of n.

n appends	Doubling: copies	Copies / n	Grow by +1: copies	Copies / n
8	7	0.88	28	3.5
64	63	0.98	2,016	31.5
1,024	1,023	1.00	523,776	511.5

Table 2.5: Total number of copies when n appends are made to an empty array. The values were produced with a counting program.

With doubling, the number of copies per append stays below 1; when n is a power of two, the number of copies is exactly $n - 1$. With growth by one, the number of copies per append is $(n - 1)/2$ and grows with n. At a thousand elements the difference is more than five hundredfold.

The amortized result does not mean that every single append is fast. The append that coincides with an enlargement still takes $O(n)$. Amortized analysis guarantees the average over a long sequence of operations; it does not rely on probability and holds for the worst sequence of operations as well. In a real-time system, where the time of a single operation matters, this distinction must be taken into account.

Doubling also has a memory cost. Right after an enlargement about half the capacity is empty. The capacity never exceeds twice the size; that is, in the worst case half the cells are unused. Some libraries use a smaller factor such as 1.5 to reduce this margin. As long as the factor stays above 1, the

total number of copies is still *O(n)*; only the constant changes.

> **Common mistake**
>
> Writing `a = realloc(a, new_size * sizeof *a);`. If `realloc` fails, it returns `NULL` and this value is written into `a`. The old block is still allocated, but its address is no longer in any variable: the block leaks and the data in it is lost. The result must first be taken into a temporary variable and, if it is not `NULL`, assigned to the real pointer.

In an experiment made for these notes, `malloc` and `realloc` were replaced with a wrapper that keeps counters, and the third `realloc` was made to fail on purpose. `append` returned 0, the array stayed intact with its two elements (0, 1), and after `free` no allocated block was left.

2.3.4 A two-dimensional dynamic array: array2d()

For a matrix whose number of rows and columns is known only at run time, the most common method is to build an **array of pointers**. First an array holding one pointer for each row is allocated, then each row is allocated as a separate block and its address is written into this array. The address of the matrix is the address of the pointer array. The type of this address is `int **`, that is, a **pointer to pointer**.

```
int **array2d(int rows, int cols) {
  int i, **x = malloc(rows * sizeof *x);
  if (x == NULL) return NULL;
  for (i = 0; i < rows; i++) {
    x[i] = malloc(cols * sizeof **x);
    if (x[i] == NULL) {                /* roll back */
      while (--i >= 0) free(x[i]);
      free(x); return NULL;
    }
  }
  return x;
}
```

The code works as follows, line by line:

- `malloc(rows * sizeof *x)` allocates a block of `rows` pointers. Since `x` is an `int **`, `*x` is an `int *` and `sizeof *x` gives the size of a pointer. If this block cannot be allocated, the function immediately returns `NULL`.
- The loop allocates `cols` `int`s for each row. Since `**x` is an `int`, `sizeof **x` is the size of an integer. The two `sizeof` expressions are different and must not be confused.

- If a row cannot be allocated, everything allocated so far is given back. `while (--i >= 0)` first decrements `i` and then checks; thus the failed row `i` is skipped and rows *i − 1, …, 0* are released. Last, the pointer array is given back and `NULL` is returned. If this step is not written, the rows already allocated leak on failure.
- If all rows were allocated, `x` is returned. The caller can use the notation `x[i][j]` as in an ordinary two-dimensional array.

Step	Operation	Result
1	`x = malloc(3 * sizeof(int *))`	pointer array of 3 cells; contents indeterminate
2	`x == NULL` ?	no, continue
3	`i = 0` : `x[0] = malloc(4 * sizeof(int))`	`x[0]` points to a block of 4 `int` s
4	`i = 1` : `x[1] = malloc(…)`	`x[1]` points to the second block
5	`i = 2` : `x[2] = malloc(…)`	`x[2]` points to the third block
6	`i = 3`, `3 < 3` false	the loop ends, `x` is returned

Table 2.6: Trace of the call `array2d(3, 4)` .

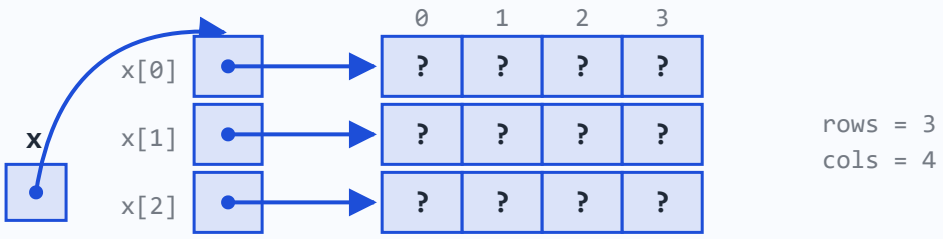


Figure 2.3: The structure built by `array2d(3, 4)` . `x` points to an array of three pointers; each pointer points to a separate row block of four `int` s. The contents of the rows are still indeterminate (?).

Figure 2.3 gives the structure as it appears in memory. The expression `x[i][j]` is resolved in two steps: `x[i]` , that is `*(x + i)` , fetches the address of row *i*; `[j]` advances *j* elements from this address. Written out in full it is `*(*(x + i) + j)` . In the driver program, the values `x[2][3] = 23` and `*(*(x + 1) + 2) = 12` were read as expected after *10i + j* was written into the rows.

There is an important difference between this structure and a real two-dimensional array such as `int a[3][4];` . In a real array all twelve elements sit in a single contiguous block. In the pointer array, each row is allocated by a separate `malloc` call, so rows need not be contiguous in memory. Two consequences follow. First, rows can have different lengths; for a lower triangular matrix it is possible to allocate only *i + 1* cells for row *i*. Second, the structure involves one more step of indirection, and it is not possible to copy the whole matrix with a single `memcpy` .

The order matters when such a matrix is given back. First every row `x[i]` and then the pointer array `x` is `free` d. If the order is reversed, `x[i]` is read after `x` has been released; that is an access to memory that has been given back. If only `free(x)` is written, the addresses of the rows are lost and all of them leak.

Common mistake

Releasing a two-dimensional dynamic array with only `free(x)`. In a matrix of three rows this leaks three blocks. The correct order: `for (i = 0; i < rows; i++) free(x[i]); free(x);`. In the experiment with counters written for these notes, the cases in which `array2d` fails at each allocation were tried one by one; in every case the function returned `NULL` and no allocated block was left behind.

In terms of cost, `array2d` makes *rows + 1* calls to `malloc`. If the time of each call is taken as constant, the function takes *O(rows)* time. The memory allocated is the sum of *rows · cols* integers and *rows* pointers. Access to the elements is *O(1)*, even though it involves two indirect steps.

2.4 Multidimensional arrays and address computation

A table, a picture, or a game board is expressed naturally with two indices; a volume or a video with three. An array whose elements are reached with more than one index is called a **multidimensional array**. Memory, however, is one-dimensional: every byte has a single address. This section examines how a multidimensional array is laid out in one-dimensional memory and how the address of an element is computed.

2.4.1 Memory is one-dimensional

An n-dimensional array declared as $A[u_0][u_1]\ldots[u_{n-1}]$ has $u_0 \cdot u_1 \cdot \ldots \cdot u_{n-1}$ elements. For example `A[10][10][10]` holds a thousand elements. The compiler must put these elements into memory in some order. There are two basic orders.

Definition 2.2

In **row-major order**, all elements of row 0 are placed first, then the elements of row 1; the rightmost index varies fastest. In **column-major order**, all elements of column 0 are placed first, then the elements of column 1; the leftmost index varies fastest.

C, C++, and the NumPy library in Python use row-major order by default. Fortran, MATLAB, and R are column-major. In a program that passes matrices between two languages, overlooking this difference results in the matrix being transposed without anyone noticing.

Whichever order is chosen, the address computation rests on the same question: *how many elements are there in memory before the element A[i][j]?* If this number is called k, the address of the element is $\alpha + k \cdot s$. Here α is the starting address of the array and s is the size of one element in bytes. In a one-dimensional array $k = i$; in a multidimensional array k is the sum of several terms.

In C, the declaration `int x[3][4];` is really an array of arrays: three elements, each an array of four `int`s. The expression `x[1]` points to the start of row 1, that is, of an array of four elements. In the driver program the difference `x[1] - x[0]` came out as 4 elements: there is exactly one row between two row starts. This structure is the row-major order showing up directly in the type system of the language.

2.4.2 Row-major and column-major formulas

Let us take the array `int x[3][4]`; the number of rows is $u_0 = 3$ and the row length is $u_1 = 4$. In row-major order there are i full rows before $x[i][j]$. Since each row has u_1 elements, these rows come to $i \cdot u_1$ elements. In the element's own row there are j elements to its left. The total:

$$\text{row-major: } \&x[i][j] = \alpha + (i \cdot u_1 + j) \cdot s$$

In column-major order, j full columns come first. Each column has u_0 elements; these come to $j \cdot u_0$ elements. Within its column, the element has i elements above it:

$$\text{column-major: } \&x[i][j] = \alpha + (j \cdot u_0 + i) \cdot s$$

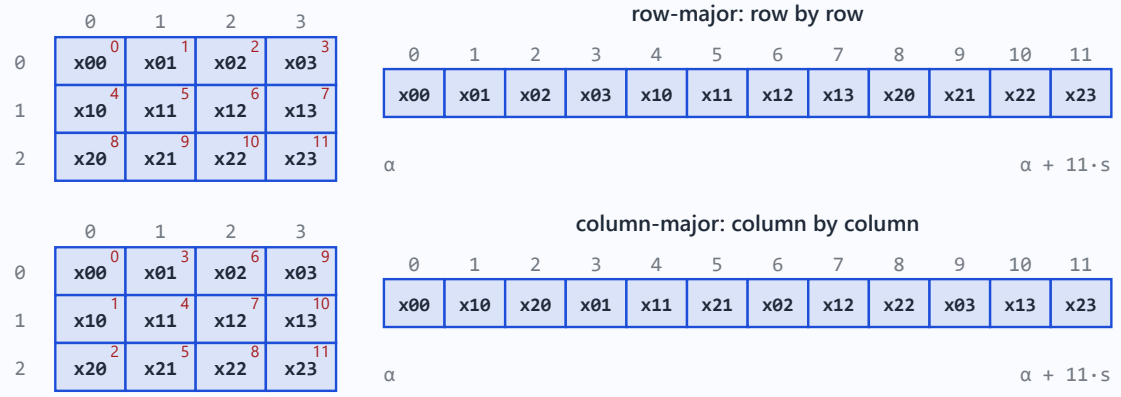


Figure 2.4: The two layouts of the array `int x[3][4]`. The small numbers in the corners of the matrix cells are the position of the element in the linear array. At the top, row-major order: the four elements of row 0 come first. At the bottom, column-major order: the three elements of column 0 come first.

Figure 2.4 shows that the same element falls at different positions in the two orders. x_{12} is at position $1 \cdot 4 + 2 = 6$ in row-major order and at position $2 \cdot 3 + 1 = 7$ in column-major order. The two formulas follow the same logic; only which index is "outer" and which is "inner" changes.

The expression in parentheses in the formula is in units of elements. To get to a byte address, it is multiplied by the element size s. Since pointer arithmetic in C does this multiplication itself, `&x[0][0] + 6` is already $6 \cdot s$ bytes ahead. When computing a byte address by hand, however, s must not be forgotten.

Common mistake

Using the number of rows (u_0) in the row-major formula. The space taken by i full rows depends on the row *length* (u_1, that is, the number of columns). The number of rows does not appear in the formula at all. In square matrices the two values are equal, so the mistake stays hidden; in a non-square matrix such as 3×4 it immediately gives a wrong address.

2.4.3 Code trace: address()

The row-major formula can be written as a small function. The function takes the starting address, the two indices, the row length, and the element size, and returns the byte address.

```c
/* int x[3][4]: row-major byte address */
long address(long alpha, int i, int j,
             int dim1, int s) {
  long k = (long)i * dim1 + j;   /* elements before */
  return alpha + k * s;
}

int main(void) {
  printf("%ld\n", address(100, 1, 2, 4, 4));
  printf("%ld\n", address(100, 2, 3, 4, 4));
  return 0;
}
```

`k` is the number of elements before the element. The cast `(long)i` makes the multiplication happen in type `long`; for very large arrays an `int` product could overflow. The function returns `alpha + k * s`. `main` asks for the addresses of two elements: *x[1][2]* and *x[2][3]*; the array starts at address 100, $u_1 = 4$, and $s = 4$.

Call	k = i · dim1 + j	Return alpha + k · s	Output
`address(100, 1, 2, 4, 4)`	$1 \cdot 4 + 2 = 6$	$100 + 6 \cdot 4 = 124$	124
`address(100, 2, 3, 4, 4)`	$2 \cdot 4 + 3 = 11$	$100 + 11 \cdot 4 = 144$	144

Table 2.7: The two calls of the function `address` ($\alpha = 100$, $u_1 = 4$, $s = 4$).

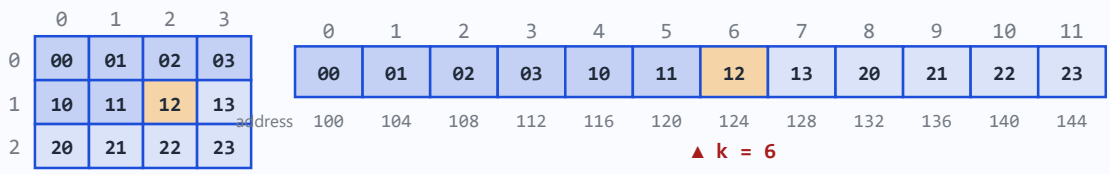


Figure 2.5: The address of *x[1][2]*. Before the element there is one full row (4 elements) and 2 elements in its own row; $k = 6$, address $100 + 6 \cdot 4 = 124$.

x[2][3] in the second call is the last element of the array. *k = 11*, that is, the last position in an array of twelve elements. When the program is compiled and run, it prints 124 and 144. Had column-major order been used, $k = 2 \cdot 3 + 1 = 7$ for *x[1][2]* and the address would have been 128. In the driver program, in a real `int x[3][4]` array the difference `&x[1][2] - &x[0][0]` also came out as 6 elements; thus one sees directly that C is row-major. The function contains no loop that depends on the input; its time is *O(1)*.

2.4.4 Three-dimensional arrays

A three-dimensional array is an array of two-dimensional arrays. The array `G[2][2][3]` has two **slices**; each slice is a matrix of 2 rows and 3 columns. In row-major order the six elements of slice 0 come first, then the six elements of slice 1. The inside of a slice is also laid out row by row.

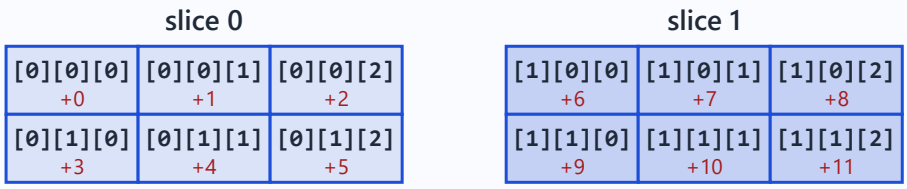


Figure 2.6: The two slices of the array `G[2][2][3]`. In each cell the indices of the element are written, and below them its linear position (*+0* … *+11*).

The elements before *G[i][j][k]* are again counted layer by layer. First there are *i* full slices; each slice comes to $u_1 \cdot u_2$ elements. Then within its own slice there are *j* full rows, each of u_2 elements. Finally in its own row there are *k* elements. Table 2.8 shows step by step how these terms are added; the addresses are in units of elements.

Element	Address
[0][0][0]	α
[i][0][0]	$\alpha + i \cdot u_1 \cdot u_2$
[i][j][0]	$\alpha + i \cdot u_1 \cdot u_2 + j \cdot u_2$
[i][j][k]	$\alpha + i \cdot u_1 \cdot u_2 + j \cdot u_2 + k$

Table 2.8: How the address of an element in a three-dimensional array is built up (in element units).

The expression in the last row has three multiplications and three additions. Since multiplication is a more expensive operation than addition, the compiler rewrites the expression by factoring out the common factor:

$$\alpha + (i \cdot u_1 + j) \cdot u_2 + k$$

In this form two multiplications and three additions remain. For `G[1][0][2]` the computation gives *(1 · 2 + 0) · 3 + 2 = 8*. Comparing with Figure 2.6, one sees that under the cell *[1][0][2]* indeed *+8* is written. In the driver program, in a real `int G[2][2][3]` array the difference `&G[1][0][2] - &G[0][0][0]` also came out as 8.

2.4.5 The n-dimensional generalization

The same idea extends to any number of dimensions. In the array $A[u_0][u_1]...[u_{n-1}]$, before $A[i_0][i_1]...[i_{n-1}]$ there are, for the outermost index, i_0 full "slices", each of $u_1\ u_2\ ...\ u_{n-1}$ elements. For the next index inward there are i_1 blocks of $u_2\ ...\ u_{n-1}$ elements. The innermost index adds only i_{n-1} elements.

Element	Address
$A[0][0]...[0]$	α
$A[i_0][0]...[0]$	$\alpha + i_0 \cdot u_1\ u_2\ ...\ u_{n-1}$
$A[i_0][i_1]...[0]$	previous row + $i_1 \cdot u_2\ ...\ u_{n-1}$
$A[i_0]...[i_{n-1}]$	previous row + i_{n-1}

Table 2.9: Address terms in an *n*-dimensional array (in element units).

The terms can be written as a sum. The coefficient of the *j*th index is the product of all the dimensions that come after it:

$$\&A[i_0]...[i_{n-1}] = \alpha + \Sigma_{j=0}^{n-1}\, i_j \cdot a_j,\quad a_j = u_{j+1} \cdot u_{j+2} \cdot ... \cdot u_{n-1}\ (0 \le j < n-1),\quad a_{n-1} = 1$$

The coefficients a_j depend only on the dimensions, not on the indices. The compiler computes them once, at compile time. A notable consequence of the formula is that u_0 appears in no coefficient. The length of the outermost dimension does not enter the address computation. This is why a function parameter in C can be written `int a[][4]` but cannot be written `int a[3][]`. The first dimension is unnecessary for the address, while the following ones are mandatory.

Computing each term separately takes many multiplications. **Horner's method** is a method that evaluates a polynomial with nested products, and it is useful here as well. The address term can be rewritten as follows:

$$(...((i_0 \cdot u_1 + i_1) \cdot u_2 + i_2) \cdot u_3 + ...) \cdot u_{n-1} + i_{n-1}$$

The computation starts with $t = i_0$ and for each dimension $t = t \cdot u_j + i_j$ is performed. The loop runs $n - 1$ times, with one multiplication and one addition per round. In total $n - 1$ multiplications suffice. The "factoring out" in the three-dimensional array is the special case of this method for $n = 3$.

2.4.6 Code trace: offset()

Horner's method is written as a general function that takes the array of indices and the array of dimensions. The function returns the relative position in element units; the byte address is found with `alpha + t * s`.

```
/* idx: indices, dim: dimensions, n: number of dimensions */
int offset(const int idx[], const int dim[], int n) {
  int j, t = idx[0];
  for (j = 1; j < n; j++)
    t = t * dim[j] + idx[j];            /* Horner */
  return t;                /* address = alpha + t*s */
}
```

The `const` in the parameters documents that the function will not modify the arrays. `t` starts with the outermost index. Since the loop starts at `j = 1`, `dim[0]` is never read. Thus the fact that the outermost dimension does not enter the address computation is visible in the code as well.

Step	`j`	Operation	`t`	Meaning
start	—	`t = idx[0]`	1	slice number 1
1	1	$t = 1 \cdot 2 + 0$	2	the "global row" number counted across slices (slice 1, row 0)
2	2	$t = 2 \cdot 3 + 2$	8	there are 8 elements before the element
end	3	`3 < 3` false	8	the loop ends, 8 is returned

Table 2.10: Position of *G[1][0][2]* in the array `G[2][2][3]` with `offset` (`idx = {1, 0, 2}`, `dim = {2, 2, 3}`, `n = 3`).

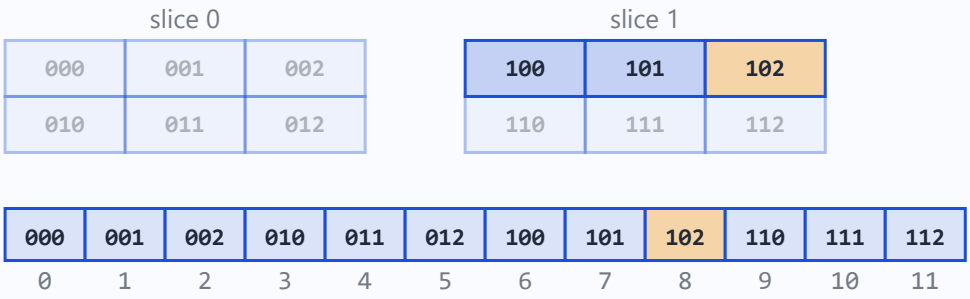


Figure 2.7: The result of `offset`. *G[1][0][2]* (orange) is cell number 8 of the linear array.

The meaning of the intermediate value $t = 2$ is interesting. If the rows of the two slices are counted one after the other (rows 0 and 1 of slice 0, rows 2 and 3 of slice 1), the sought element is in row number 2. In the last step, multiplying this row number by the row length (3) and adding the column index gives the position of the element. The computation uses two multiplications in all, that is, $n - 1$. The direct formula gives the same result: $1 \cdot 2 \cdot 3 + 0 \cdot 3 + 2 = 8$. In the driver program the relative position of *[2][3][4]* was also tried for an array `A[4][5][6]`, and the result of the function (82) came

out the same as the difference in the real array. The time of `offset` is linear in the number of dimensions, *Θ(n)*. Since the number of dimensions is fixed for an array, access is *O(1)* in terms of the number of elements.

2.5 Triangular matrices

The address formula is useful not only to the compiler. The programmer can also arrange memory and find the place of elements with a formula. The simplest application of this idea is to store matrices that are half empty by keeping only their filled halves.

2.5.1 The lower triangular matrix

A square matrix is divided into two by its main diagonal. In some applications one of the halves suffices. In a table of distances between cities, the distance from *i* to *j* is the same as the distance from *j* to *i*; half the table is a repeat of the other half. In solving systems of linear equations, too, matrices that are zero above the diagonal occur often.

Definition 2.3

A **lower triangular matrix** is a square matrix all of whose elements above the main diagonal are zero: if $i < j$ then $a_{ij} = 0$.

There is no need to store the zeros. The rest of the matrix is kept row by row in a one-dimensional array `LOW`. The user, however, is offered an interface as if working with a two-dimensional matrix: when a_{ij} is requested, the function computes its position in `LOW`.

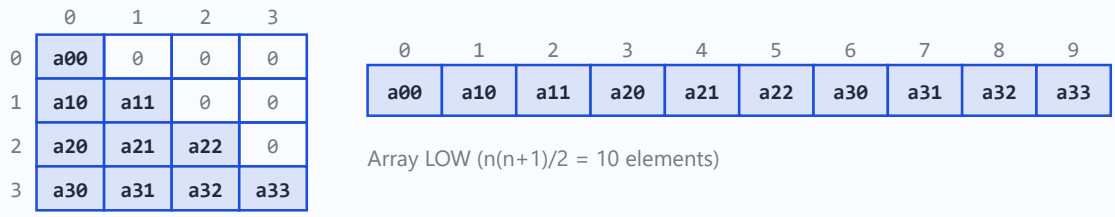


Figure 2.8: A *4 × 4* lower triangular matrix and the 10-element array `LOW` that holds it row by row. The six zeros above the diagonal are not stored.

To find the formula, the same question is asked again: how many elements are there in `LOW` before a_{ij}? In row *i*, *i + 1* elements are stored (from column 0 to *i*). The previous rows come to *1 + 2 + ... + i = i(i + 1)/2* elements. In its own row the element has *j* elements to its left:

$$k = i(i + 1)/2 + j \quad (j \le i)$$

In an $n \times n$ lower triangular matrix, $1 + 2 + \ldots + n = n(n + 1)/2$ elements are stored in all. A full matrix needs n^2 cells. The saving is $n(n - 1)/2$ cells, which for large n means about half the memory. The formula can be tried on Figure 2.8: for a_{21} one finds $k = 2 \cdot 3/2 + 1 = 4$, and `LOW[4]` is indeed a_{21}.

2.5.2 Code trace: lower_read()

The following function reads the stored elements of an $n \times n$ lower triangular matrix row by row from standard input and places them in the array `low`.

```
#define MAX_SIZE 100
void lower_read(int low[], int n) {
  int i, j, k;
  if (n * (n + 1) / 2 > MAX_SIZE) {
    printf("Array too small!\n"); exit(1);
  }
  for (i = 0; i < n; i++) {
    k = (i + 1) * i / 2;      /* elements before */
    for (j = 0; j <= i; j++)  /* j <= i */
      scanf("%d", &low[k + j]);
  }
}
```

The code works as follows, line by line:

- First the number of elements to be stored, $n(n + 1)/2$, is compared with the capacity of the array, `MAX_SIZE`. If it does not fit, a message is printed and the program ends. Without this check a large n would overrun the end of the array.
- The outer loop walks over the rows. `k = (i + 1) * i / 2` is the number of elements already placed before row i; that is, the start of the row in `low`. The division is done after the multiplication; since $i(i + 1)$ is always even, the division is exact.
- The inner loop runs only for `j <= i`; the zeros above the diagonal are not read. Each value is written to `low[k + j]`.

Example 2.2

Let the input for $n = 4$ be `3 1 4 1 5 9 2 6 5 3`. First the check $4 \cdot 5/2 = 10 \le 100$ passes. Table 2.11 shows where each value is written.

`i`	`k`	`j`	Position `k + j`	Value read	Matrix element
0	$1 \cdot 0/2 = 0$	0	0	3	a_{00}
1	$2 \cdot 1/2 = 1$	0	1	1	a_{10}
1	1	1	2	4	a_{11}

`i`	`k`	`j`	Position `k + j`	Value read	Matrix element
2	*3 · 2/2 = 3*	0	3	1	a_{20}
2	3	1	4	5	a_{21}
2	3	2	5	9	a_{22}
3	*4 · 3/2 = 6*	0	6	2	a_{30}
3	6	1	7	6	a_{31}
3	6	2	8	5	a_{32}
3	6	3	9	3	a_{33}

Table 2.11: Trace of the call `lower_read(low, 4)`.

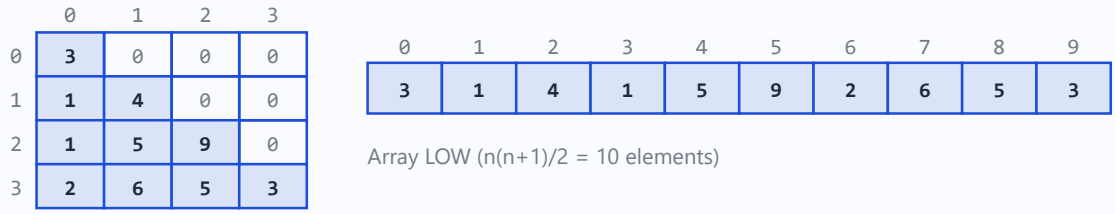


Figure 2.9: The matrix and the array `LOW` when reading is finished. Ten values were read; the six zeros in the upper triangle were never stored.

The body of the function performs one `scanf` for each stored element. The total number of rounds of the inner loop is *1 + 2 + ... + n = n(n + 1)/2*. The time is proportional to the number of stored elements, $\Theta(n^2)$. Reading a full matrix would take n^2 rounds; the triangular representation halves the constant factor but does not change the growth rate. In the driver program the array `low` was printed after reading as `3 1 4 1 5 9 2 6 5 3` and the table was verified.

2.5.3 Access and error codes: lower_get()

To access the matrix after reading, a function is needed that converts the pair *(i, j)* into a position inside `LOW`. The function must tell apart two exceptions: the indices may be outside the matrix, or the requested element may be in the upper triangle, that is, in the zero region that is not stored.

```
int lower_get(int i, int j, int n) {
  if (i < 0 || i >= n || j < 0 || j >= n) {
    printf("Invalid index\n");
    return -2;
  } else if (i < j) return -1;    /* upper triangle: 0 */
  else return (i + 1) * i / 2 + j;   /* valid */
}
/* main: */
printf("%d\n", lower_get(2, 1, 4));
printf("%d\n", lower_get(1, 3, 4));
printf("%d\n", lower_get(4, 0, 4));
```

The function first checks the bounds. If one of the indices is outside the range *[0, n − 1]*, it prints a message and returns −2. Then it looks at whether *i < j*. If so, the element is in the upper triangle; its value is zero but it has no place in `LOW`, and the function returns −1. In the remaining case the element is in the lower triangle and the position is computed with the formula.

Call	Bounds check	`i < j` ?	Return	Output
`lower_get(2, 1, 4)`, a_{21}	valid	2 < 1, no	$3 \cdot 2/2 + 1 = 4$	4
`lower_get(1, 3, 4)`, a_{13}	valid	1 < 3, yes	−1 (upper triangle)	-1
`lower_get(4, 0, 4)`, a_{40}	$i = 4 \geq n$, invalid	—	−2	Invalid index, -2

Table 2.12: The three calls of the function `lower_get` ($n = 4$).

The output of the driver program consists of the lines `4`, `-1`, `Invalid index`, and `-2`. The choice of return values is deliberate. Valid positions are nonnegative numbers such as 0, 1, 2, ... The error codes {−1, −2} do not intersect this set. The caller can therefore always tell the situation apart by looking at the return value: a negative value is an exception, and zero or a positive value is a real position. Had 0 been chosen as an error code, for example, it would be confused with the position of a_{00}.

The function contains no loop that depends on the input: it makes a few comparisons and one formula evaluation. Its time is *O(1)*. It is as fast as the access `a[i][j]` in a full matrix, yet works with half the memory.

Common mistake

Writing the position formula as `i * i / 2 + j` or `(i + 1) * (i + 2) / 2 + j` instead of `i * (i + 1) / 2 + j`. The first gives the row start too small (2 instead of 3 at $i = 2$), the second gives the start of the next row. The formula is easily tested on a small matrix with the corner elements: $a_{00} \to 0$, $a_{10} \to 1$, $a_{n-1,n-1} \to n(n + 1)/2 - 1$.

2.5.4 Other triangles and the symmetric matrix

The stored half can take four forms, depending on which side of the diagonal it lies. In each, the position formula is derived by the same method: the elements in the previous rows are added up, then the order within the row is added. Table 2.13 gives the result for the four cases.

Stored region	Elements in row *i*	Position *k*
$j \leq i$ (lower left)	$i + 1$	$i(i + 1)/2 + j$
$j \geq i$ (upper right)	$n - i$	$i \cdot n - i(i - 1)/2 + (j - i)$
$i + j \leq n - 1$ (upper left)	$n - i$	$i \cdot n - i(i - 1)/2 + j$
$i + j \geq n - 1$ (lower right)	$i + 1$	$i(i + 1)/2 + j - (n - 1 - i)$

Table 2.13: Row-major position formula for the four kinds of triangles ($n \times n$ matrix).

In the upper right triangle there are $n - r$ elements in row r. The total before row i is $n + (n - 1) + \ldots + (n - i + 1) = i \cdot n - i(i - 1)/2$. The first stored element of the row is on the diagonal ($j = i$), so the order within the row is $j - i$. In the upper left triangle the row lengths are the same but the row starts at column 0; the order is directly j. In the lower right triangle row i holds $i + 1$ elements and starts at column $j = n - 1 - i$; the order is $j - (n - 1 - i)$. A loop written for these notes tried the four formulas on every matrix from $n = 1$ to 8. The positions given by the formulas always came out the same as the positions counted row by row.

In a real **symmetric matrix** ($a_{ij} = a_{ji}$) the upper triangle is not zero but the mirror image of the lower triangle. In this case `lower_get`, instead of returning −1 for the upper triangle, swaps the indices: if $i < j$ then $k = j(j + 1)/2 + i$. A distance table or the undirected adjacency matrix of a graph can be stored this way. The memory again halves and no information is lost.

2.6 Band matrices

In the triangular matrix the stored region was one side of the diagonal. In many numerical problems, however, the nonzero elements gather in a narrow strip on both sides of the diagonal. In the numerical solution of physics problems such as heat conduction or vibration, each unknown is related only to its neighbors; the elements of the resulting matrix far from the diagonal are zero. Storing such a matrix in full means spending most of the memory on zeros.

2.6.1 Definition and element count

Definition 2.4

A **band matrix** is a square matrix whose meaningful elements gather in a strip around the main diagonal. The elements outside the strip are unused or all have the same constant value c (most often $c = 0$). The matrix is described by three numbers: the side length n, the number a of lower diagonals including the main diagonal, and the number b of upper diagonals including the main diagonal.

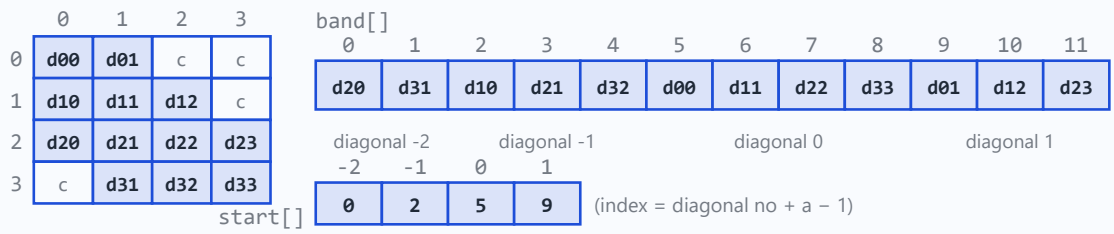


Figure 2.10: A band matrix with $n = 4$, $a = 3$, $b = 2$. On the left the matrix (elements outside the band are c); on the right the array `band`, which holds the elements diagonal by diagonal, and the array `start`, which holds the start of each diagonal.

In the matrix of Figure 2.10, the main diagonal and the two diagonals below it ($a = 3$) and one diagonal above the main diagonal ($b = 2$, main diagonal included) are stored. d_{02}, d_{03}, d_{13}, and d_{30} fall outside the band.

To find the number of stored elements, the diagonals are counted one by one. The main diagonal has n elements. The diagonal one step away (below or above) has $n - 1$, and the one two steps away has $n - 2$ elements. In general a diagonal at distance t from the main diagonal has $n - |t|$ elements. In all $a + b - 1$ diagonals are stored, because the main diagonal has been counted both in a and in b. If every diagonal had n elements, the total would be $n(a + b - 1)$. The shortfalls in the lower diagonals are $1 + 2 + \ldots + (a - 1) = a(a - 1)/2$, and in the upper diagonals $b(b - 1)/2$:

$$\text{number of elements} = n(a + b - 1) - a(a - 1)/2 - b(b - 1)/2$$

In the example $4 \cdot 4 - 3 - 1 = 12$ elements are stored; a full matrix would need 16 cells. The saving looks small, because the matrix is small. In a matrix with $n = 1000$ and $a = b = 3$, however, the full matrix needs a million cells while the band representation makes do with $1000 \cdot 5 - 3 - 3 = 4994$ cells.

The storage problem is thus solved. The real question is whether the stored elements can still be reached quickly. When the elements are compressed, the position computation becomes more complicated too. The value of the band matrix representation depends on access still being possible in constant time.

2.6.2 Diagonal number and the START array

The key to access is to tell from its indices at once which diagonal an element lies in. For elements in the same diagonal the difference $j - i$ is constant. For example d_{20} and d_{31} are in the same diagonal: $0 - 2 = 1 - 3 = -2$. This difference is called the **diagonal number**. The number of the main diagonal is 0; negative numbers denote the lower diagonals and positive numbers the upper ones.

The valid diagonal numbers lie in the closed interval $[1 - a, b - 1]$. In the example this interval is $[-2, 1]$. Elements whose number falls outside this interval are in the constant region and are not stored. Thus whether an element is inside or outside the band is found out with a single subtraction and two comparisons.

The elements are stored in the array `band` diagonal by diagonal, starting from the lowest diagonal. The position of an element has two parts: the start of its diagonal inside `band` and the order of the element within the diagonal.

- **Start of the diagonal.** The start of each diagonal is kept in the array `start` (START). The array has $a + b - 1$ elements, that is, one per diagonal. Since an array index in C cannot be negative, the diagonal number is used shifted by $a - 1$: the start of diagonal number t is at `start[t + a - 1]`. In the example `start = {0, 2, 5, 9}`; diagonal −2 starts at 0, diagonal −1 at 2, the main diagonal at 5, and diagonal +1 at 9.

- **Order within the diagonal.** The first element of a lower diagonal is in column 0; along the diagonal the column increases by one at each step. So in a lower diagonal the order is j. The first element of an upper diagonal is in row 0; the order is i. On the main diagonal $i = j$, so both give the same result.

The thought behind the design is this: examine everything as if it were constrained, and derive the access mechanism from those constraints. That the diagonal number is constant, that the diagonal lengths are $n - |t|$, and that the place of the first element is known are three constraints. Together they reduce an apparently complicated compression to a few arithmetic operations.

2.6.3 Code trace: band_build()

The following function reads the elements of the band matrix from standard input. The input comes diagonal by diagonal, starting from the lowest diagonal. The function writes the start of each diagonal into `start` and places the elements in `band`. The function `abs` is in the header `<stdlib.h>`.

```
void band_build(int band[], int start[],
                int n, int a, int b) {
  int i, k, total = 0;   /* previous diagonals */
  if (n*(a+b-1) - a*(a-1)/2 - b*(b-1)/2 > MAX_SIZE) {
    printf("Out of memory!\n"); exit(-1);
  }
  for (i = 1 - a; i <= b - 1; i++) {  /* diagonal by diagonal */
    start[i + a - 1] = total;       /* diagonal start */
    for (k = 0; k <= n - abs(i) - 1; k++)
      scanf("%d", &band[start[i + a - 1] + k]);
    total += n - abs(i);
  }
}
```

The code works as follows, line by line:

- `total` is the number of elements placed so far; that is, the start of the next diagonal.
- First the number of elements to be stored is compared with `MAX_SIZE`. The formula is the element count obtained from the sum of the diagonal lengths.
- The outer loop walks the diagonal number `i` from $1 - a$ to $b - 1$. Here `i` is not a row index but a diagonal number.
- `start[i + a - 1] = total` records the start of the diagonal.
- The inner loop reads the $n - |i|$ elements of the diagonal: `k` goes from 0 to $n - |i| - 1$. Each value is written to `band[start[i + a - 1] + k]`.
- When the diagonal ends, `total` grows by the length of the diagonal.

Example 2.3

For $n = 4$, $a = 3$, $b = 2$, let the values of the matrix be $d_{00} = 1$, $d_{01} = 2$, $d_{10} = 3$, $d_{11} = 4$, $d_{12} = 5$, $d_{20} = 6$, $d_{21} = 7$, $d_{22} = 8$, $d_{23} = 9$, $d_{31} = 10$, $d_{32} = 11$, $d_{33} = 12$. The input given diagonal by diagonal is `6 10 · 3 7 11 · 1 4 8 12 · 2 5 9`. The check passes with $4 \cdot 4 - 3 - 1 = 12 \le$ `MAX_SIZE`. Table 2.14 gives the processing of each diagonal.

Diagonal `i`	Length $n - \lvert i \rvert$	`start[i + a - 1]`	Positions written ← value (element)	`total` after
−2 (lower)	2	`start[0] = 0`	0 ← 6 (d_{20}), 1 ← 10 (d_{31})	2
−1 (lower)	3	`start[1] = 2`	2 ← 3 (d_{10}), 3 ← 7 (d_{21}), 4 ← 11 (d_{32})	5
0 (main)	4	`start[2] = 5`	5 ← 1 (d_{00}), 6 ← 4 (d_{11}), 7 ← 8 (d_{22}), 8 ← 12 (d_{33})	9
1 (upper)	3	`start[3] = 9`	9 ← 2 (d_{01}), 10 ← 5 (d_{12}), 11 ← 9 (d_{23})	12

Table 2.14: Trace of the call `band_build(band, start, 4, 3, 2)`.

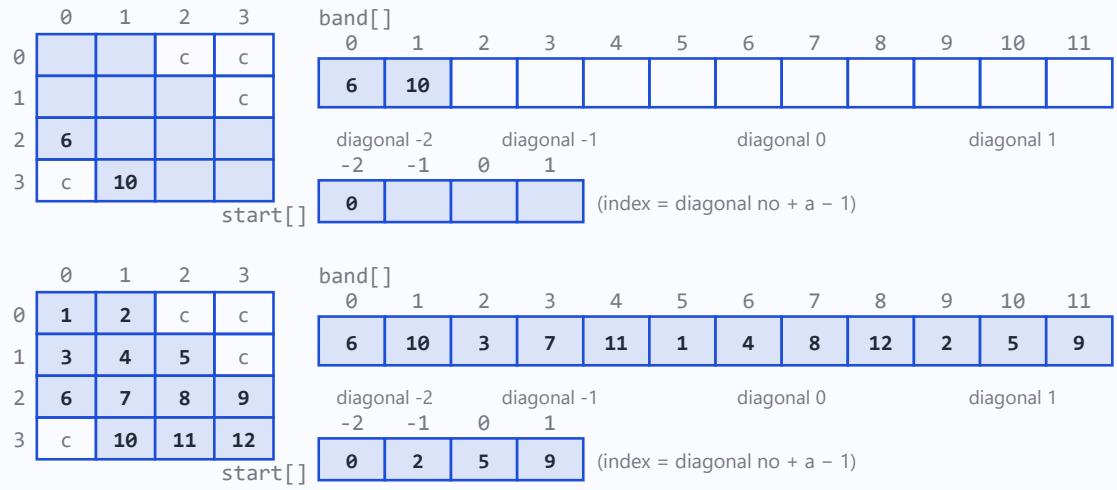


Figure 2.11: Two moments of the construction. (a) The first diagonal (−2) has been placed: `band[0] = 6`, `band[1] = 10`, `start[0] = 0`. (b) All diagonals have been placed: `band = {6, 10, 3, 7, 11, 1, 4, 8, 12, 2, 5, 9}`, `start = {0, 2, 5, 9}`.

In the end `total` reaches 12; this is the value given by the element count formula. The arrays `start` and `band` printed by the driver program are the same as in the table. The function performs one `scanf` for each stored element, and the outer loop runs $a + b - 1$ times. The time is $O(a + b + \text{number of stored elements})$; since the number of diagonals cannot exceed the number of elements, this is $O(\text{number of stored elements})$. The extra memory is the $a + b - 1$ cells of the array `start`.

2.6.4 Code trace: band_get()

The access function converts the pair (i, j) into a position inside `band`. For elements outside the band it returns two separate error codes, so that the upper and lower constant regions can be told apart.

```
int band_get(int i, int j, int n, int a,
             int b, int start[]) {
  if (i < 0 || j < 0 || i >= n || j >= n) {
    printf("Index overflow\n"); return -3;
  } else if (j - i >= 1) {         /* upper triangle */
    if (j - i < b)               /* valid diagonal */
      return start[j - i + a - 1] + i;
    else return -1;              /* upper constant region */
  } else if (i - j < a)          /* main / lower */
    return start[j - i + a - 1] + j;
  else
    return -2;                   /* lower constant region */
}
```

The logic of the function divides into four cases:

- If one of the indices is outside the matrix, a message is printed and −3 is returned.
- If $j - i \geq 1$, the element is above the diagonal. If the diagonal number is less than b, the element is in a valid upper diagonal and the position is `start[j - i + a - 1] + i`. Otherwise the element is in the upper constant region; −1 is returned.
- Otherwise the element is on the main diagonal or below it. If $i - j < a$, it is in a valid lower diagonal (main diagonal included) and the position is `start[j - i + a - 1] + j`.
- If none of these holds, the element is in the lower constant region; −2 is returned.

Element	Diagonal no. $j - i$	Decision	Return	Value
d_{31}	−2	$i - j = 2 < 3$: lower diagonal	`start[0] + j` = 0 + 1 = 1	`band[1]` = 10
d_{12}	1	$1 \geq 1$: upper; $1 < b = 2$: valid	`start[3] + i` = 9 + 1 = 10	`band[10]` = 5
d_{03}	3	$3 \geq 1$: upper; $3 \geq b$: outside the band	−1	c
d_{30}	−3	$i - j = 3 < 3$ false: outside the band	−2	c

Table 2.15: Four queries with `band_get` ($n = 4$, $a = 3$, $b = 2$, `start = {0, 2, 5, 9}`).

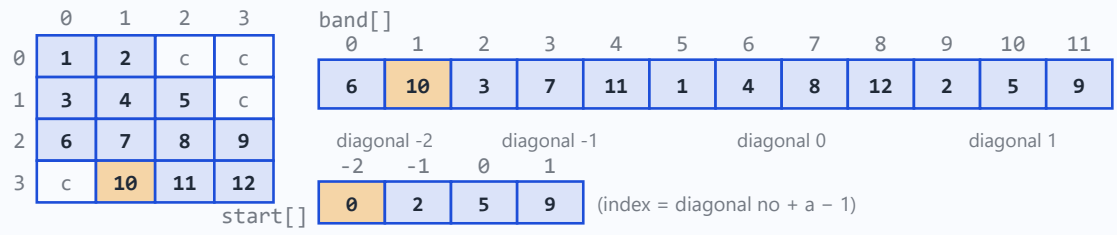


Figure 2.12: The query for d_{31}. The element is the second element of diagonal −2; its position is `start[0] + 1 = 1` and its value is 10.

In the driver program the four queries gave the same results. In addition, the access function was called for all pairs *(i, j)* and the matrix was rebuilt, with 0 written for elements outside the band. The resulting matrix is the same as the values in the example. In every case the function performs a constant number of comparisons and arithmetic operations; it has no loop that depends on the input. Its time is *O(1)*.

> **Common mistake**
>
> Computing the position in an upper diagonal as `start[…] + j`. The first element of an upper diagonal is in row 0 but not in column 0; the upper diagonal number *t* starts at column *t*. The order must be counted from the row, not from the column. For d_{12} the faulty formula gives *9 + 2 = 11* and returns the value of d_{23} (9).

2.6.5 The START array or a formula?

`band_get` runs in constant time because the diagonal starts were written beforehand into the array `start`. Without this array, every access would need a loop that adds up the lengths of the previous diagonals, and access would rise to *O(a + b)*. `start` prevents this but has a price: it uses extra memory, and if the parameters of the matrix change it must be rebuilt. That also takes the program away from being dynamic.

A general design principle can be seen here. If it really improves performance, there is no harm in using a reasonable amount of extra memory. `start` is only *a + b − 1* cells; next to the stored elements it is very small. Still, the same result can be had without any extra memory: the start of a diagonal can also be computed with a closed formula.

```
/* t: diagonal no (1-a..b-1); same as start[t+a-1] */
int diag_start(int t, int n, int a) {
  if (t <= 0)                    /* lower diagonals */
    return (t + a - 1) * n
           - (a * (a - 1) + t * (1 - t)) / 2;
  return (a - 1) * n - a * (a - 1) / 2  /* start at t=0 */
         + t * n - t * (t - 1) / 2;
}
```

The formula is derived in two cases.

- **Lower diagonals and the main diagonal (*t ≤ 0*).** Before *t* there are the *t + a − 1* diagonals numbered *1 − a, …, t − 1*. Had all of them *n* elements, the total would be *(t + a − 1) · n*. The shortfalls are the absolute values of the numbers of these diagonals: *(−t + 1) + … + (a − 1) = a(a − 1)/2 − t(t − 1)/2*. In the code this expression is written as `(a * (a - 1) + t * (1 - t)) / 2`; the two forms are the same.

- **Upper diagonals ($t > 0$).** The start of the main diagonal is found by putting $t = 0$ in the formula above: $(a - 1) \cdot n - a(a - 1)/2$. To it are added the main diagonal and the lengths of the diagonals numbered $1, \ldots, t - 1$: $n + (n - 1) + \ldots + (n - t + 1) = t \cdot n - t(t - 1)/2$.

For $n = 4$ and $a = 3$, the values $t = -2, -1, 0, 1$ give 0, 2, 5, and 9 in turn; the same as `start`. A loop written for these notes counted 4,356 diagonal starts one by one for $n \le 11$ and all $a, b \le n$. No difference with the formula appeared. Access with the formula becomes `diag_start(j - i, n, a) + (j > i ? i : j)`. This is also $O(1)$; a few extra multiplications are made but no array is needed. Which to choose depends on whether memory or running time is more valuable.

2.6.6 Evaluation order of the && operator

The conditions in the access functions are compound expressions such as `i < 0 || i >= n || …`. There is a common belief about such expressions. According to it, one cannot know whether the compiler scans the expression from left to right or from right to left. Therefore the second of the two forms below is safer.

```
/* 1 */ if (i < n && a[i] != 0) ...

/* 2 */ if (i < n)
          if (a[i] != 0) ...
```

This belief is wrong. The C standard guarantees left-to-right evaluation and **short-circuit evaluation** for the operators `&&` and `||` (C99 §6.5.13–14). If the left side of `&&` is false, the result is already false and the right side is never evaluated. In `||`, if the left side is true, the right side is not evaluated. In the first form, if `i < n` is false, `a[i]` is never read. The two forms are equally safe; the first is a common and correct pattern in C. In the driver program the first form was tried on an array of three elements with `i = n = 3`; the condition came out false without `a[3]` being read.

The real traps are elsewhere. `&` (bitwise AND) and `|` do not short-circuit; if `i < n & a[i] != 0` is written, both sides are evaluated and `a[n]` is read. Moreover, in an ordinary expression such as `f(x) + g(x)`, the order in which the operands are evaluated is not specified by the standard. If `f` and `g` modify the same global variable, the result may differ from compiler to compiler. The order of evaluation is guaranteed only for `&&`, `||`, `?:`, and the comma operator.

2.7 Structures and unions

All elements of an array have the same type. Records in the real world, however, consist of parts of different types: a student's number is an integer, the name is a character array, and the grade point average is a floating-point number. C defines such records with `struct`. A `union`, on the other hand, is a special kind of record in which several fields share the same memory. This section covers the two constructs, their appearance in memory, and the effect of alignment on size.

2.7.1 struct

Definition 2.5

A `struct` (structure) is a C type that gathers fields of different types into a single record. The fields are laid out in memory in the order of declaration. A field is accessed as `variable.field`, and through a pointer to a structure as `pointer->field`.

```c
typedef struct {
  char name[10];
  int age;
  float salary;
} person;

person p1, p2;
strcpy(p1.name, "Ali");
p1.age = 21;
p1.salary = 3500.0f;
p2 = p1;          /* copied field by field */
/* if (p1 == p2) -> DOES NOT COMPILE */
```

The code works as follows, line by line:

- `typedef struct { … } person;` defines an anonymous structure and gives it the name `person`. `typedef` is a declaration that gives a new name to an existing type. This way one does not need to write `struct` every time a variable is declared: `person p1, p2;` suffices.
- `p1.name` is a `char` array. Since arrays cannot be assigned to, the name is copied with `strcpy`. `p1.age` and `p1.salary` are assigned like ordinary variables. The `f` at the end of `3500.0f` says that the constant is a `float`.
- `p2 = p1;` copies the whole structure field by field. What is interesting is this: the `name` array inside the structure is copied as well. An array alone cannot be assigned to, but if the array is a field of a structure, the structure assignment carries it along.
- `p1 == p2` does not compile. C defines no equality operator for structures. The reason is alignment: there may be meaningless **padding bytes** between fields, and a byte-by-byte comparison could give a wrong result. To compare two structures, a function that compares field by field is written; `char` array fields are compared with `strcmp`.

Structures are often passed to functions by pointer. If a structure is passed by value, all its fields are copied; for large structures this is expensive. Passing by pointer copies only an address. After the definition `person *p = &p2;`, `p->age` and `(*p).age` give the same field. The parentheses are necessary because the operator `.` has higher precedence than `*`: `*p.age` means "what `p.age` points to" and does not compile. The **arrow operator** exists to shorten this notation.

In the driver program `sizeof(person)` came out as 20. The sum of the fields is *10 + 4 + 4 = 18* bytes. The 2 bytes in between come from alignment: the field `age` must start at an address that is a multiple of 4, and since `name` ends at byte 10, two padding bytes are added.

2.7.2 Nested and self-referential structures

A field of a structure can be another structure. A structure can also carry a pointer field that points to an object of its own type.

```
typedef struct {
  int day, month, year;
} date;

typedef struct {
  char name[10];
  date birth;           /* nested structure */
} member;
/* m.birth.year = 2004; */

typedef struct list {
  char data;
  struct list *link;    /* self-reference */
} list;
```

The field `birth` of the structure `member` is a `date` structure. Nested fields are reached by applying the dot operator repeatedly: `m.birth.year = 2004;`. In the driver program `sizeof(member)` came out as 24: the 10-byte name, 2 padding bytes, and the 12-byte date.

The structure `list` is different. The type of the field `link` is `struct list *`, that is, a pointer to the structure's own type. Such structures are called **self-referential structures**. It is mandatory to give the structure a tag name (`struct list`) in the definition. The name `list` given with `typedef` is not yet defined until the declaration ends; the type of the field can only be written with the tag.

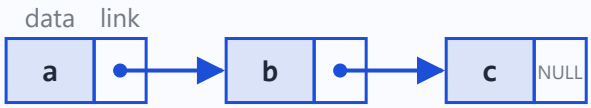


Figure 2.13: A linked list of three nodes built from self-referential structures. In the `data` field of each node sits a character, and in the `link` field the address of the next node; the link of the last node is `NULL`.

Figure 2.13 shows how three such structures are linked together. Each node holds the address of the next; the end of the chain is marked with `NULL`. In the driver program three nodes were built with the values `a`, `b`, `c` and printed in order with the loop `for (q = &a1; q != NULL; q = q->link)`.

This structure is the basis of linked lists, trees, and graphs. Unlike an array, the nodes do not have to be contiguous in memory; insertion in the middle is done by changing only a few links.

A structure cannot contain a *field* of its own type. In a definition such as `struct list { char data; struct list next; };` the size of the structure would have to include its own size; that means an infinite size and it does not compile. A *pointer* to its own type, on the other hand, causes no problem, because the size of a pointer is independent of the size of the type pointed to (8 bytes on a 64-bit system). In the driver `sizeof(list)` came out as 16: the 1-byte character, 7 padding bytes, and the 8-byte pointer.

2.7.3 union and alignment

> **Definition 2.6**
>
> A `union` is a C type all of whose fields share the same region of memory. The size of a union is that of its largest field (it may grow a little for alignment). Only one field at a time holds a valid value.

A union saves memory when a record needs to carry different information depending on its kind. In the example in the textbook, two different pieces of information are kept depending on a person's gender. Since the two fields are never needed at the same time, they can share the same memory.

```
typedef struct {
  enum { FEMALE, MALE } kind;   /* tag */
  union {
    int children;              /* if FEMALE */
    int beard;                 /* if MALE */
  } u;
} gender;

gender c;
c.kind = MALE;
c.u.beard = 0;   /* read only the field that matches the tag */
```

The type of the field `kind` is an **enumeration** type (`enum`): `FEMALE` and `MALE` are named constants taking the values 0 and 1. This field is a *tag* that shows which field of the union is valid. Such a structure is called a **tagged union**. The code sets the tag to `MALE` and then uses only the field that matches it, `beard` .

Keeping the tag correct is entirely the programmer's job. C does not track which field of the union was written last. If `c.u.children` is read after `c.u.beard = 0;` was written, the same four bytes are interpreted with another meaning. Here both fields are `int` , so the result is 0, but had the fields been of different types, a meaningless value would be read. The correct pattern is to look at the tag before

every access to the union: `if (c.kind == MALE) … c.u.beard …`. In the driver program `sizeof c.u` came out as 4 and `sizeof c` as 8 bytes: the 4-byte tag and the 4-byte union. Had two separate fields been used, the structure would be 12 bytes.

The real factor that determines structure size is **alignment**. Processors read most types fastest from addresses that are multiples of their own size; some processors do not allow unaligned reads at all. The compiler therefore places each field at a suitable address multiple and, if necessary, inserts padding bytes in between. The total size of the structure is also rounded up to a multiple of the largest alignment requirement; this way every element in arrays of structures starts at the correct alignment.

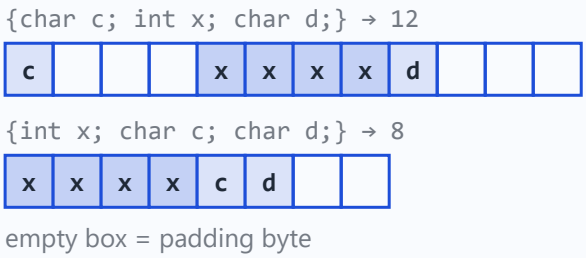


Figure 2.14: The effect of field order on structure size (4-byte `int`, a typical 32/64-bit compiler). At the top `{char c; int x; char d;}` takes 12 bytes, at the bottom `{int x; char c; char d;}` takes 8 bytes. The empty boxes are padding bytes.

Figure 2.14 compares two different orders of the same three fields. In the first order `c` sits at byte 0; because `x` must start at a multiple of 4, bytes 1, 2, 3 become padding and `x` occupies bytes 4–7. `d` sits at byte 8. The size of the structure must be a multiple of 4, so three padding bytes are added at the end: 12 in total. In the second order `x` occupies bytes 0–3, `c` byte 4, and `d` byte 5; two padding bytes are added at the end and the total is 8. The same data is held with one third less memory. In the driver program the sizes of these two structures came out as 12 and 8, and the position of `x` measured with `offsetof` came out as 4 in the first.

The practical rule is to order the fields from largest to smallest. The alignment rules depend on the compiler and the processor; the values given here are typical for common 32- and 64-bit systems. For structures written to a file or to the network as raw bytes, padding bytes cause portability problems. That is why in such cases the fields are written one by one.

2.7.4 Code trace: class_avg()

Structures can be collected in arrays. The following function takes an array holding the student records of a class and returns the average of the grade point averages. Visiting all the elements of a data structure once each in a certain order is called **traversal**.

```c
typedef struct {
  int id;
  char name[16];
  double avg;
} Student;

double class_avg(const Student s[], int n) {
  double sum = 0.0;
  int i;
  for (i = 0; i < n; i++)
    sum += s[i].avg;              /* . for a field */
  return n > 0 ? sum / n : 0.0;   /* empty class */
}
```

The code works as follows, line by line:

- The parameter `const Student s[]` is a pointer; the array is not copied. `const` documents that the function will not modify the records and makes the compiler report an error if a field is written by mistake.
- `s[i].avg` first finds record i (the address $\alpha + i \cdot$ `sizeof(Student)`), then reads its `avg` field. In an array of structures the address formula applies as is; only the element size is the size of the structure.
- `n > 0 ? sum / n : 0.0` prevents division by zero in an empty class. With floating-point numbers, division by zero does not stop the program but produces infinity or NaN; this value silently spreads to later computations.

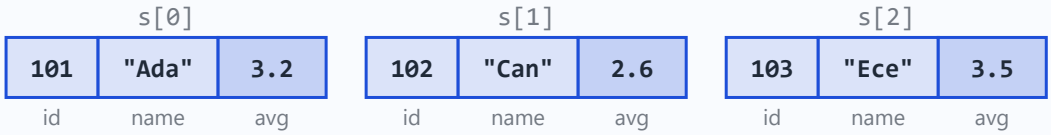


Figure 2.15: An array of structures made of three student records. The records sit one after another in memory; each record is `sizeof(Student)` bytes.

`i`	`i < n`	`s[i].avg`	`sum` **after**
0	0 < 3, true	3.2	3.2
1	1 < 3, true	2.6	5.8
2	2 < 3, true	3.5	9.3
3	3 < 3, false	—	the loop ends; $n > 0$, return $9.3 / 3 = 3.10$

Table 2.16: Trace of the call `class_avg(s, 3)` with the three records of Figure 2.15.

The function visits each record once; its time is $\Theta(n)$ and its extra memory $O(1)$. In the driver program the result was printed as 3.10 with the format `%.2f`. The exact values of the intermediate sums are 3.2000000000000002, 5.8000000000000007, and 9.3000000000000007. Decimal numbers such as 3.2 cannot be represented exactly in binary, and small rounding errors accumulate. That is why floating-point results are not compared with `==`; they are either printed with a given number of digits or compared within a small tolerance.

The structure `Student` also shows the effect of alignment. The 4-byte `id` and the 16-byte `name` come to 20 bytes. A `double` must start at a multiple of 8; so 4 padding bytes are added and `avg` starts at byte 24. The total size is 32 bytes. In the driver program `offsetof(Student, avg)` came out as 24 and `sizeof(Student)` as 32.

2.8 Strings

Text is one of the kinds of data that programs process most. Many languages offer a separate type for text. In C, text is an ordinary `char` array that obeys a special rule. This simplicity gives speed, but when used carelessly it is also the source of the most common security holes.

2.8.1 A char array terminated by '\0'

Definition 2.7

In C a **string** is a `char` array terminated by the character `'\0'`, whose value is zero. This character is called the **null character**. The length of the string is the number of characters before the first `'\0'`.

```
char s[] = "dog";      /* 4 bytes: d o g \0 */
char t[10];
char *p = "dog";       /* read-only constant */

strlen(s)   /* 3 : up to the \0 */
sizeof s    /* 4 : size of the array */

strcpy(t, s);                /* is t big enough? */
snprintf(t, sizeof t, "%s!", s);   /* safe */
if (strcmp(s, "dog") == 0) ...     /* not == ! */
```



Figure 2.16: The declaration `char s[] = "dog";` in memory: three characters and the terminating `'\0'`, 4 bytes in all.

The code works as follows, line by line:

- `char s[] = "dog";` allocates an array of four bytes. The compiler copies the string and adds `'\0'` at the end; its size is therefore 4, not 3 (Figure 2.16).
- `char t[10];` is an empty area of ten characters; its contents are as yet indeterminate.
- `char *p = "dog";` does not allocate an array. `p` points to a string constant in the read-only region of the program. Writing `p[0] = 'D';` is undefined behavior and on most systems crashes the program. That is why such pointers should be declared `const char *p`; then the compiler catches an attempt to write.
- `strlen(s)` gives 3 and `sizeof s` gives 4. `strlen` counts characters up to the `'\0'`; `sizeof` is the memory the array occupies.
- `strcpy(t, s)` copies `s` to `t` together with its terminator. `strcpy` does not know the size of the destination; if `t` is not big enough, it overruns the end. Here 10 bytes are enough for a string of 4 bytes.
- `snprintf(t, sizeof t, "%s!", s)` takes the size of the destination as its second parameter, writes at most `sizeof t - 1` characters, and always puts a `'\0'` at the end. Overflow is impossible. In the driver the new value of `t` was `dog!`.
- `strcmp(s, "dog") == 0` compares the contents of two strings character by character. Had `s == "dog"` been written, addresses and not contents would be compared and the result would most likely be wrong.

The header `<string.h>` offers a set of functions for strings. Table 2.17 summarizes the most used ones. All these functions recognize the end of a string by the `'\0'`. If an array without a terminator is given to these functions, they keep reading until they find an accidental zero further on in memory.

Function	Job	Time (*n*: string length)
`strlen`	length (excluding `'\0'`)	*O(n)*
`strcpy` / `strncpy`	copy	*O(n)*
`strcat`	append	*O(n + m)*; the end of the destination is searched for first

Function	Job	Time (*n*: string length)
`strcmp`	compare in dictionary order; 0 if equal, negative if the first is smaller, positive if larger	*O(n)*
`strchr` / `strstr`	search for a character / substring; `NULL` if not found	*O(n)* / *O(n · m)* in a simple implementation

Table 2.17: Frequently used `<string.h>` functions.

`strncpy` has a trap. If the source is longer than the given limit, `strncpy` does not put a `'\0'` in the destination; the result is not a string. For a safe copy, therefore, `snprintf` is preferred, or `strncpy` with the limit kept one smaller and a `'\0'` written by hand afterwards.

Common mistake

Not reserving room for the terminator in a string. `char name[3] = "Ali";` compiles in C (it does not in C++), but the array holds only three characters and the `'\0'` is left outside. `strlen(name)` or `printf("%s", name)` keeps reading past the end of the array. A string of *n* characters always needs *n + 1* bytes. The same holds for dynamic allocation: `malloc(strlen(s) + 1)`.

2.8.2 Code trace: length()

The following function writes out explicitly what `strlen` does. It advances a pointer until it reaches the `'\0'` and returns how far it has moved from the start.

```c
size_t length(const char *s) {
  const char *p = s;
  while (*p != '\0')     /* up to the terminator */
    p++;
  return (size_t)(p - s);  /* pointer difference */
}

/* length("data") */
```

`p` initially points to the same place as `s`. In each round the loop compares `*p` with the terminator; if different, it advances `p` by one character (1 byte). When the loop ends, `p` points to the terminator. The difference of the two pointers, `p - s`, is the number of elements between them; that is the length of the string. The difference is of type `ptrdiff_t` and is returned converted to `size_t`.

Round	`p - s`	`*p`	`*p != '\0'`	Action
1	0	`'d'`	true	`p++`
2	1	`'a'`	true	`p++`
3	2	`'t'`	true	`p++`

Round	`p - s`	`*p`	`*p != '\0'`	Action
4	3	`'a'`	true	`p++`
5	4	`'\0'`	false	the loop ends; return 4

Table 2.18: Trace of the call `length("data")`.

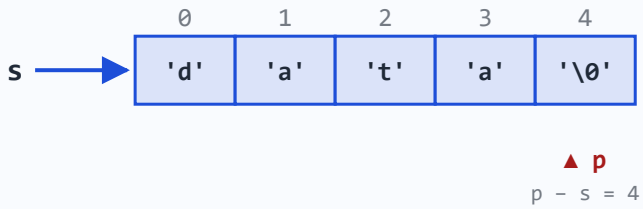


Figure 2.17: When the loop ends, `p` points to the terminator; $p - s = 4$.

To find the length the string is scanned from beginning to end: the time is $\Theta(n)$. Unlike structures such as arrays, whose length is kept separately, the length of a C string is recomputed every time. This leads to a commonly seen inefficiency:

```
for (i = 0; i < strlen(s); i++)   /* strlen on every round! */
    ...
```

The loop condition is recomputed in every round, and each computation scans n characters. For n rounds a total of $n \cdot n$ characters are read; the loop becomes $O(n^2)$. In a text of a million characters this means a trillion reads instead of a million. The right way is to compute the length once before the loop: `size_t len = strlen(s); for (i = 0; i < len; i++)`. If the loop body does not modify the string, good compilers may do this transformation themselves, but one should not rely on it.

2.9 Matrix operations

Two-dimensional arrays are used most in matrix computations. This section examines matrix addition and multiplication; in particular it derives the exact step count of multiplication. In the examples `N` is a constant defined at compile time, and matrices are passed in the form `int a[][N]`. Writing the second dimension is mandatory, because the address computation needs the row length.

2.9.1 Matrix addition

In the sum of two matrices of the same size, each cell is the sum of the cells at the same position in the two matrices. For $m \times N$ matrices the code consists of two nested loops.

```
void add(int a[][N], int b[][N], int c[][N], int m) {
    int i, j;
    for (i = 0; i < m; i++)
        for (j = 0; j < N; j++)
            c[i][j] = a[i][j] + b[i][j];
}
```

The body of the inner loop runs once for each cell, $m \cdot N$ times in all. The loop tests are also proportional to this number. The time is $\Theta(mN)$; for $n \times n$ square matrices, $\Theta(n^2)$. No extra memory is needed apart from the result matrix. Since each cell is read once and written once, there is no faster addition algorithm: every cell of the input must be looked at at least once. In the driver the sum of two 2×3 matrices was tried and the expected result was obtained.

2.9.2 Code trace: mult()

Let A and B be two $n \times n$ matrices. Each cell of the product $C = AB$ is the inner product of a row of A with a column of B:

$$c_{ij} = \Sigma_{k=0}^{n-1} a_{ik} \cdot b_{kj}$$

```
#define N 2
void mult(int a[][N], int b[][N], int c[][N]) {
  int i, j, k;
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++) {
      c[i][j] = 0;
      for (k = 0; k < N; k++)
        c[i][j] += a[i][k] * b[k][j];
    }
}
```

The outer two loops walk over the cells of the result matrix. Each cell is first zeroed, then the innermost loop adds up n products. For $A = [[1, 2], [3, 4]]$ and $B = [[5, 6], [7, 8]]$ the trace is given in Table 2.19.

i	j	k	Operation	c[i][j] after
0	0	0	$0 + a[0][0] \cdot b[0][0] = 1 \cdot 5$	5
0	0	1	$5 + a[0][1] \cdot b[1][0] = 5 + 2 \cdot 7$	19
0	1	0	$0 + a[0][0] \cdot b[0][1] = 1 \cdot 6$	6
0	1	1	$6 + a[0][1] \cdot b[1][1] = 6 + 2 \cdot 8$	22
1	0	0	$0 + a[1][0] \cdot b[0][0] = 3 \cdot 5$	15
1	0	1	$15 + a[1][1] \cdot b[1][0] = 15 + 4 \cdot 7$	43

`i`	`j`	`k`	**Operation**	`c[i][j]` **after**
1	1	0	*0 + a[1][0] · b[0][1] = 3 · 6*	18
1	1	1	*18 + a[1][1] · b[1][1] = 18 + 4 · 8*	50

Table 2.19: Trace of the function `mult` for *N = 2*.



Figure 2.18: The product of *2 × 2* matrices: *C = [[19, 22], [43, 50]]*. The innermost line ran $N^3 = 8$ times.

The driver program printed the result as `[[19, 22], [43, 50]]`. The innermost assignment ran eight times, that is, N^3 times. As n grows, this number grows as a cube; when the matrix size doubles, the work grows eightfold.

2.9.3 Step count, rectangular matrices, and the transpose

The exact step count of the multiplication is found by counting how many times each line runs. The condition of a `for` loop is tested one more time than the body runs: the last test comes out false and the loop ends. Table 2.20 gives the count line by line.

Line	**Times executed**
`for (i …)` condition	$n + 1$
`for (j …)` condition	$n(n + 1)$
`c[i][j] = 0`	n^2
`for (k …)` condition	$n^2(n + 1)$
`c[i][j] += …`	n^3
Total	$2n^3 + 3n^2 + 2n + 1$

Table 2.20: Step count of `mult` on $n \times n$ matrices.

The total is found by adding the terms $(n + 1) + (n^2 + n) + n^2 + (n^3 + n^2) + n^3$. The dominant term is $2n^3$; the time is $\Theta(n^3)$. The number of multiplications performed is exactly n^3. In a version with a counter written for these notes, the steps were counted for $n = 1$ to 9 and came out the same as the formula (for example 33 at $n = 2$ and 88 at $n = 3$).

Matrices need not be square. The product of an $m \times n$ matrix and an $n \times p$ matrix has size $m \times p$; that is, it has $m \cdot p$ cells. Each cell is the sum of n products. The total number of multiplications and the time are $\Theta(m \cdot n \cdot p)$. This result also explains why the order of multiplication matters in a chain of

matrices. *(AB)C* and *A(BC)* give the same result but may need very different numbers of multiplications.

The **transpose** of a matrix is the matrix with its rows and columns exchanged: a_{ij} and a_{ji} are swapped. For a square matrix this can be done without an extra array, **in place**.

```c
void transpose(int a[][N], int n) {
    int i, j, t;
    for (i = 0; i < n; i++)
        for (j = i + 1; j < n; j++) { /* upper triangle only */
            t = a[i][j];
            a[i][j] = a[j][i];
            a[j][i] = t;
        }
}
```

The inner loop starts at `j = i + 1`; only the cells above the diagonal are visited. Each swap exchanges an upper triangle cell with its counterpart in the lower triangle. Had the loop started at `j = 0`, every pair would be swapped twice: the first swap takes the transpose and the second reverses it, and the matrix would end up unchanged. The diagonal cells stay in place. The number of swaps is $(n - 1) + (n - 2) + \ldots + 0 = n(n - 1)/2$; the time is $\Theta(n^2)$ and the extra memory is a single temporary variable. In the driver a 3×3 matrix was transposed with three swaps. Transposing a non-square matrix in place is much harder, because the row length changes and the elements move around the array in cycles.

2.10 Complexity summary

Table 2.21 gives the time and extra memory costs of the operations in the chapter together. The times are for the worst case; amortized ones are noted separately.

Operation	Time	Extra memory
Access *A[i]*; *n*-dimensional address ($n - 1$ multiplications)	$O(1)$	$O(1)$
Append to a dynamic array (doubling)	amortized $O(1)$, worst case $O(n)$	capacity $\le 2 \cdot$ size
Insertion / deletion in the middle	$O(n)$	$O(1)$
Access to a lower triangular / band matrix	$O(1)$	$n(n + 1)/2$ / band elements
`band_build`, `lower_read`	$O($stored elements$)$	START: $a + b - 1$
`strlen` / `length`	$O(n)$	$O(1)$
Matrix addition ($m \times n$)	$O(mn)$	$O(1)$

Operation	Time	Extra memory
Matrix multiplication ($n \times n$)	$O(n^3)$	$O(1)$

Table 2.21: Complexity of the operations in the chapter.

The table has a common lesson: when the layout in memory is known, access reduces to a formula and is done in constant time. In one-dimensional arrays, multidimensional arrays, and triangular and band matrices alike, the same question has been asked: "how many elements are there before the sought element?" The operations that raise the cost are those that move elements: insertion in the middle, growing, and scanning.

2.11 Common mistakes

Most of the mistakes in this chapter arise from a deliberate choice of C: the language does not check indices, the result of memory allocation, or the end of a string. In the list below each mistake is given together with its correct pattern.

1. **Writing outside the array bounds.** Accessing `a[n]` is undefined behavior and C does not check it. The last index of an array of n elements is $n - 1$; the loop condition is written `i < n`.
2. **Writing `a = realloc(a, …)`.** On failure the address of the old block is lost and the block leaks. The result is first taken into a temporary variable and, if it is not `NULL`, assigned to the real pointer.
3. **Confusing `sizeof(int *)` with `sizeof(int)`; using `sizeof a` on a parameter.** The first allocates memory of the wrong size. In the second, since the parameter is a pointer, the size of the pointer comes out, not that of the array. The pattern `malloc(n * sizeof *a)` and passing the length as a separate parameter prevent these two mistakes.
4. **Releasing only the pointer array of a two-dimensional dynamic array without `free`ing the rows.** All the rows leak. First every `x[i]` and then `x` is released.
5. **Confusing the row-major and column-major formulas; forgetting the element size (s).** The row-major formula uses the row length (u_1); for a byte address the element count is multiplied by s.
6. **Using `==` on strings or not reserving room for `'\0'`.** `==` compares addresses; use `strcmp` for contents. A string of n characters needs $n + 1$ bytes.
7. **Not taking alignment into account.** The size of a structure can be larger than the sum of its fields. The size is always obtained with `sizeof`, never added up by hand.

2.12 Summary

- At the abstract level an **array** is a **mapping** made of ⟨index, value⟩ pairs; its operations are `Create`, `Retrieve`, and `Store`. In the C implementation the elements sit contiguously and the address of *A[i]* is computed as $\alpha + i \cdot s$ in $O(1)$. Insertion and deletion in the middle need shifting and are therefore $O(n)$.
- C does not check indices; accessing `a[n]` is **undefined behavior**. An array name turns into the address of the first element in most expressions: `list[i]` ≡ `*(list + i)`. **Pointer arithmetic** counts elements, not bytes. An array name cannot be assigned to; an array passed to a function turns into a pointer, and its length is sent separately.
- `malloc` allocates a block from heap memory; its return is checked for `NULL`, the block is `free`d when the work is done, and `NULL` is assigned to the pointer. The pattern `sizeof *a` gives the right size even when the type changes.
- A **dynamic array doubles** its capacity when it is full. In n appends the total number of copies is less than $2n$; the **amortized** cost per append is $O(1)$. Growing by a fixed increment means $O(n^2)$ copies. The result of `realloc` is first taken into a temporary variable.
- A two-dimensional dynamic array consists of an **array of pointers** and separate row blocks (`int **`). The rows need not be contiguous; when releasing, first the rows and then the pointer array are `free`d.
- C uses **row-major** order: $\&x[i][j] = \alpha + (i \cdot u_1 + j) \cdot s$. In column-major order it is $\alpha + (j \cdot u_0 + i) \cdot s$. In an n-dimensional array the address is $\alpha + \Sigma\ i_j\ a_j$; **Horner's method** computes it with $n - 1$ multiplications. The outermost dimension does not enter the address computation.
- A **lower triangular matrix** is stored with $n(n + 1)/2$ elements; the position of a_{ij} is $i(i + 1)/2 + j$. The access function returns error codes (−1, −2) that do not intersect the valid positions.
- In a **band matrix** the difference $j - i$ of elements on the same diagonal is constant. The number of elements is $n(a + b - 1) - a(a - 1)/2 - b(b - 1)/2$. The diagonal starts are kept in the START array or computed with a closed formula; with either route access is $O(1)$.
- `&&` and `||` are evaluated left to right and with **short-circuiting**; `i < n && a[i] != 0` is safe. `&` and `|` do not short-circuit.
- `struct` gathers fields of different types; it can be assigned but cannot be compared with `==`. A **self-referential structure** is the basis of linked structures. The fields of a `union` share the same memory; the valid field is tracked with a tag. **Alignment** adds padding bytes; a structure can be made smaller by ordering the fields from largest to smallest.
- A C **string** is a `char` array terminated by `'\0'`; n characters need $n + 1$ bytes. `strlen` is $O(n)$; if it is called in a loop condition, the loop becomes $O(n^2)$. Contents are compared with `strcmp`; `snprintf` is used against overflow.
- Matrix addition is $\Theta(mn)$, multiplication of square matrices is $\Theta(n^3)$ with $2n^3 + 3n^2 + 2n + 1$ steps, and multiplication of rectangular matrices is $\Theta(mnp)$. The in-place transpose of a square matrix takes $n(n - 1)/2$ swaps.

2.13 Key terms

abstract data type (ADT) implementation array mapping index pointer pointer arithmetic pointer to pointer null pointer (NULL) dangling pointer sizeof dynamic memory allocation heap memory malloc / calloc / realloc / free memory leak dynamic array capacity doubling amortized analysis multidimensional array row-major order column-major order Horner's method lower triangular matrix band matrix short-circuit evaluation struct typedef arrow operator (->) self-referential struct union (C) tagged union enumeration (enum) alignment traversal string null character ('\0') transpose in place

2.14 Test Yourself

1. The array `int a[6];` starts at address 1000 and `sizeof(int) = 4`. What address do `&a[4]` and `a + 4` give? Which element is `*(a + 4)`?

   Answer

   Both give the address *1000 + 4 · 4 = 1016*. Since pointer arithmetic counts elements, `a + 4` is 4 elements ahead, that is 16 bytes, not 4 bytes. `*(a + 4)` is `a[4]` itself.

2. The array `short s[4][7];` starts at address 500 and `sizeof(short) = 2`. What is `&s[2][5]` in row-major order? What would it be if the array were stored column-major?

   Answer

   Row-major: before the element there are 2 full rows (*2 · 7 = 14* elements) and 5 elements in its row; *500 + 19 · 2 = 538*. Column-major: first 5 full columns (*5 · 4 = 20* elements) and 2 elements in its column; *500 + 22 · 2 = 544*. The row-major formula uses the row length (7), and the column-major formula the column length (4).

3. In the array `int A[3][4][5];`, what is the relative position of *A[2][1][3]* found with Horner's method? How many multiplications are made in the computation?

   Answer

   t = 2; *t = 2 · 4 + 1 = 9*; *t = 9 · 5 + 3 = 48*. Two multiplications are made (*n − 1*, *n = 3*). The first dimension, 3, does not enter the computation at all. In a real C array `&A[2][1][3] - &A[0][0][0]` is also 48.

4. A *6 × 6* lower triangular matrix is stored in an array `LOW`. How many elements does the array

have? At which position is a_{52}? What does `lower_get(2, 5, 6)` return?

Answer

$6 \cdot 7/2 = 21$ elements. a_{52}: $5 \cdot 6/2 + 2 = 17$. `lower_get(2, 5, 6)` asks for a_{25} in the upper triangle; since $2 < 5$, −1 is returned (the value is 0 and is not stored).

5. How many elements does a band matrix with $n = 5$, $a = 2$, $b = 2$ (a tridiagonal matrix) hold? What is the START array? At which positions of the array `band` are d_{34} and d_{43}?

Answer

$5 \cdot 3 - 1 - 1 = 13$ elements. The diagonals are −1, 0, 1 with lengths 4, 5, 4; START = [0, 4, 9]. d_{34}: diagonal $4 - 3 = 1$, an upper diagonal; position $start[1 + 1] + i = 9 + 3 = 12$. d_{43}: diagonal −1, a lower diagonal; position $start[0] + j = 0 + 3 = 3$.

6. If 20 elements are appended to a dynamic array that starts empty and doubles its capacity, how many copies are made in total and what is the final capacity? What would the number of copies be if the capacity were increased by 1 each time?

Answer

Enlargements happen when 1, 2, 4, 8, and 16 elements fill the array: $1 + 2 + 4 + 8 + 16 = 31$ copies, final capacity 32. This is less than $2 \cdot 20 = 40$. With growth by 1, $0 + 1 + \ldots + 19 = 190$ copies are made.

7. How many bytes does `struct { char a; int b; char c; double d; }` take on a typical 64-bit system? To what size can it be reduced by reordering the fields?

Answer

`a` at byte 0, 3 padding, `b` at 4–7, `c` at byte 8, 7 padding, `d` at 16–23: 24 bytes in total. In the order `{ double d; int b; char a, c; }` the $8 + 4 + 1 + 1 = 14$ bytes get 2 bytes of padding and the size becomes 16. Both values were measured in the driver program.

8. About how many characters does the loop `for (i = 0; i < strlen(s); i++) if (s[i] == ' ') count++;` read on a string of 10,000 characters? How is the loop fixed?

Answer

The condition is recomputed in every round and each `strlen` scans 10,000 characters. In 10,001 tests about 10^8 characters are read; the loop is $O(n^2)$. The length is computed once before the loop (`size_t len = strlen(s);`), or the loop is written with the condition `s[i] != '\0'` ; both are $O(n)$.

9. A 2×3 matrix is multiplied by a 3×4 matrix. How many cells does the result have and how many multiplications are made?

   Answer

   The result is 2×4, that is, it has 8 cells. Each cell is the sum of 3 products: $2 \cdot 3 \cdot 4 = 24$ multiplications.

10. What does the function `size_t f(int a[10]) { return sizeof a; }` return on a 64-bit system? Why?

    Answer

    It returns 8. Even though the parameter is written as `int a[10]`, the compiler treats it as `int *a`; `sizeof a` is the size of the pointer, not 40. The length of the array must be sent to the function as a separate parameter. Compilers issue a warning for this usage.

2.15 Further reading

- E. Horowitz, S. Sahni, and S. Anderson-Freed, *Fundamentals of Data Structures in C* (2nd ed.), Chapter 2: the array ADT, structures, polynomial and sparse matrix representations, address computation for multidimensional arrays, and strings; this is the textbook these notes follow.
- B. W. Kernighan and D. M. Ritchie, *The C Programming Language* (2nd ed.), Chapters 5–6: the relationship between pointers and arrays, arrays of pointers, multidimensional arrays, and structures, in the words of the designers of the language.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms* (3rd ed.), Section 17.4: analyzes the expansion and contraction of dynamic tables in an amortized way with the aggregate, accounting, and potential methods.

2.16 References

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 3rd ed., MIT Press, 2009.
2. E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2nd ed., Silicon Press, 2008.
3. B. W. Kernighan, D. M. Ritchie, *The C Programming Language*, 2nd ed., Prentice Hall, 1988.

PART I · CHAPTER 3

Recursion, Binary Search and Backtracking

Some problems contain a smaller copy of themselves. The factorial of a number rests on the factorial of one less. Searching a sorted array reduces to searching half of it. Moving three disks requires knowing how to move two. These notes describe the recursive algorithms that solve such problems. They first examine how recursion works on the machine, that is, the call stack and the stack frame. Then they write binary search with a loop and with recursion, derive the recurrence relation from the towers of Hanoi, and generate permutations. The last part is devoted to the backtracking method: subset generation, satisfiability (SAT), n-queens, and the knight's tour are solved with it. Every piece of code is given in C, traced step by step, and its time and memory costs are computed with justification.

LEARNING OUTCOMES

Think recursively
Express a problem as a base case and a shrinking subproblem; compare recursion with a loop.

Follow the stack
Draw the stack frame of each call (parameters, local variables, return address) and compute the memory cost of recursion.

Binary search
Write iterative and recursive binary search in C, trace it on an array, and show the $O(\log n)$ bound.

Recurrence relations
Set up and solve the recurrence $h(n) = 2h(n - 1) + 1$ for the towers of Hanoi; show that permutation generation is $O(n \cdot n!)$.

Backtracking
Tell brute force and backtracking apart; solve subset generation, SAT, and n-queens with these methods.

Heuristics
Explain why Warnsdorff's rule for the knight's tour is much faster than backtracking.

3.1 Recursive algorithms

Someone new to programming learns first the loop as a way to repeat a job. A loop runs the same body again and again as long as a condition holds. Recursion achieves repetition another way: a function calls itself to do a small part of the job. This idea, which looks odd at first sight, is very old in mathematics. Definitions and proofs by induction have the same structure.

3.1.1 What is recursion?

If the solution logic used for the whole of a problem also holds for its parts, a recursive algorithm can be written. The factorial is the plainest example. The identity $n! = n \cdot (n-1)!$ ties the factorial of n to the factorial of a smaller number. $(n-1)!$ is tied to $(n-2)!$ by the same rule. The chain stops at $1! = 1$.

Definition 3.1

Recursion is a function calling itself with a smaller input. A function that calls itself is called **recursive**. If a function calls itself directly, there is **direct recursion**: `fact` → `fact`. If a function calls another function, which at the end of the chain calls the first function again, there is **indirect recursion**: `A` → `B` → `A`.

A familiar example of indirect recursion is expression parsers. The function `expr()`, which parses an arithmetic expression, calls the function `term()` to read terms. When a new expression begins inside parentheses, `term()` calls `expr()` again. The two functions never call themselves directly, but the chain forms a closed cycle. When writing indirect recursion in C, the declaration (prototype) of the second function must be given before the first function; otherwise the compiler does not recognize a name it has not yet seen.

3.1.2 Base case and recursive step

A function that calls itself must stop somewhere. The factorial chain stops at $1! = 1$, because this value is known without referring to another factorial. Every recursive algorithm consists of these two parts.

Definition 3.2

The **base case** is the smallest instance, solved directly without resorting to recursion. The **recursive step** reduces the problem to a smaller instance of the same kind and builds the real solution from the solution of that instance. Every recursive call must come a little closer to the base case.

The last sentence of the definition is essential for the correctness of recursion. The word "smaller" is expressed with a measure: n in the factorial, the number of elements in the range in binary search, the number of disks in the towers of Hanoi. The measure must decrease at every call and reach its smallest value in the base case. If the measure does not decrease, the calls go on forever. On a real machine "forever" does not last long, because every call takes up space in memory and the allocated memory runs out quickly.

Two questions are asked when writing a recursive function. First, which is the smallest instance and what is its answer? Second, if the answer to an instance one size smaller were known, how would the real answer be built? The second question is the essence of recursive thinking. One does not think about how the smaller instance is solved; one takes it as solved. This assumption is the same as the "let it hold for $n - 1$" assumption in a proof by induction.

> **Common mistake**
>
> Forgetting the base case or writing a step that never reaches it. The function `long f(int n) { return n * f(n - 1); }` has no base case; the calls go past $n = 0$ and continue with negative numbers, and the program crashes with a **stack overflow**. If a base case is written but the step does not approach it, the result is the same: the base case `if (n == 1) return 1;` never catches the call `f(0)`. In the factorial the condition `n <= 1` is safer because it covers both 0 and 1.

Beginners often ask a question: does the recursive solution not use a lot of memory? To answer it, one must first be able to write recursive code, and then to compute the complexity of that code. A judgment of "it uses a lot of memory" made without having learned to write recursive code remains baseless. That is also why these notes are ordered this way: code first, measurement afterwards. The memory cost turns into a concrete number once the call stack is examined.

3.1.3 A first example: factorial

The recursive definition of the factorial transfers to C almost one to one:

```c
long fact(int n) {
  if (n <= 1)
    return 1;              /* base case */
  return n * fact(n - 1);
}

int main(void) {
  printf("%ld\n", fact(3));
  return 0;
}
```

The function returns a value of type `long`, because the factorial grows very fast. The `if (n <= 1)` in the first line catches the base case and returns 1. The last line is the recursive step: the expression `n * fact(n - 1)` first has `fact(n - 1)` computed and multiplies the returned value by `n`. The function `main` calls `fact(3)` and prints the result with the format `%ld`; the output is 6.

The reading of the code is the same as the reading of the definition: "if *n* is at most 1 the factorial is 1, otherwise it is *n* times the factorial of one less." A version with a loop does not have this sentence; in its place there is a counter and an accumulating variable. The biggest gain of recursion is this readability.

The correctness of the code is shown by induction. For $n \le 1$ the function returns 1 and $0! = 1! = 1$. For $n > 1$, if `fact(n - 1)` is assumed to return $(n-1)!$, the function returns $n \cdot (n-1)! = n!$. The base case and the step together give correctness for every $n \ge 0$.

The size of the type `long` varies with the system, and this difference shows up immediately in the factorial. On 64-bit Linux and macOS `long` is 64 bits; its largest value is $2^{63} - 1 \approx 9.2 \cdot 10^{18}$ and $20! \approx 2.4 \cdot 10^{18}$ fits within this limit. On Windows, however, `long` is 32 bits; its largest value is $2^{31} - 1 = 2{,}147{,}483{,}647$. $12! = 479{,}001{,}600$ fits, $13! = 6{,}227{,}020{,}800$ does not. Signed integer overflow is **undefined behavior** in C: the program may print a wrong number or behave unexpectedly. When the code of these notes was compiled with the option `-fsanitize=undefined` and tried on Windows, the call `fact(13)` reported exactly this error. For large factorials, `long long` (at least 64 bits) or big-number libraries are used.

3.2 The call stack

Recursion is a definition on paper, but on a computer it works through a real mechanism. Understanding this mechanism is necessary both for tracing recursive code and for computing its memory cost. The mechanism is called the call stack, and it is used at every function call, recursive or not.

3.2.1 The stack frame

When a function is called, control does not pass to the body of the function at once. First some preparation is done. The **return address**, that is, where the program will go back to when the function ends, is saved. The values of the parameters are copied to a place the function can reach. Room is allocated for the function's local variables. All this information is gathered in a single record that sits consecutively in memory.

Definition 3.3

A **stack frame** (activation record) is the record that holds the parameters, local variables, and return address of a function call, together with a pointer to the previous frame. The frames of the active calls sit on top of one another, in the order of calling, in a region of memory called the **call stack**.

As its name suggests, the call stack is a stack structure: last in, first out. When a function is called, its frame is added to the top of the stack. When the function ends with `return`, its frame is removed from the top and control passes to the return address in the frame. When the program starts, only the frame of `main` is on the stack. This first frame too holds a return address and a pointer to the previous frame; when `main` ends, control returns to the operating system.

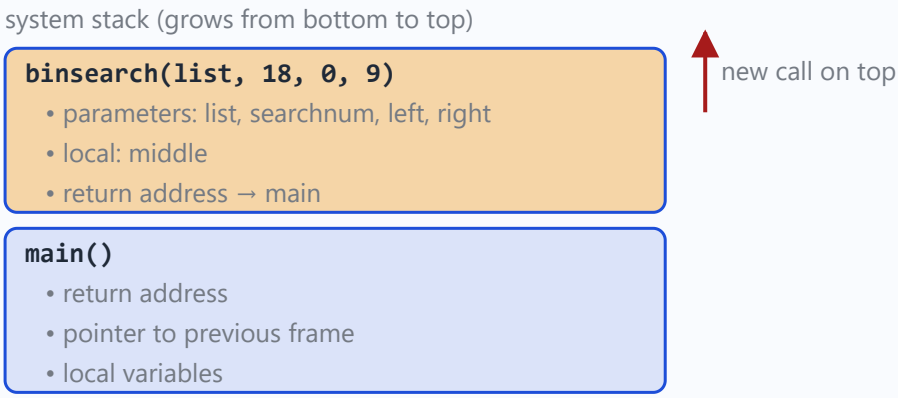


Figure 3.1: The system stack when `main` calls `binsearch(list, 18, 0, 9)`. The frame of the new call is added on top; it holds the parameters, the local variable `middle`, and the return address to `main`.

Figure 3.1 shows this arrangement. In the figure the stack grows from bottom to top; the bottom frame is that of `main`. When `binsearch` is called, its frame is added on top. In this frame there are four parameters (`list`, `searchnum`, `left`, `right`), one local variable (`middle`), and the return address to `main`. One must note that the parameter `list` is not the array itself. In C an array parameter turns into a pointer carrying the address of the array's first element. It is this address that sits in the frame; the ten elements of the array stay in the frame of `main` and are not copied.

The frame layout of a real compiler varies with the processor and the compiler. Some parameters may be carried in processor registers instead of memory. Still, the logic is the same everywhere: every active call has its own frame, and these frames are kept in a stack order. This logic is enough for understanding recursion.

Recursion has no special case. When a function calls itself, a new frame is opened as well. Several calls of the same function can be on the stack at once, and each has its own copies of the parameters and local variables. The `n` in one call is entirely separate from the `n` in another.

3.2.2 Code trace: fact(3)

How the call `fact(3)` changes the stack can be followed step by step. Table 3.1 gives, at each step, the line being executed, the current value of `n`, and the contents of the stack. The stack is written from left to right; the rightmost frame is the top of the stack.

Step	Line executed	n	Stack (bottom to top)	Event
1	`printf(…, fact(3))`	—	main	`main` calls `fact(3)`.
2	`if (n <= 1)`	3	main | fact(3)	*3 > 1*, not the base case.
3	`return n * fact(n - 1);`	3	main | fact(3)	The multiplication waits; `fact(2)` is called first.
4	`if (n <= 1)`	2	main | fact(3) | fact(2)	*2 > 1*, not the base case.
5	`return n * fact(n - 1);`	2	main | fact(3) | fact(2)	The multiplication waits; `fact(1)` is called first.
6	`if (n <= 1) return 1;`	1	main | fact(3) | fact(2) | fact(1)	Base case: 1 is returned. The stack is at its deepest.
7	`return n * fact(n - 1);`	2	main | fact(3) | fact(2)	*2 · 1 = 2* is returned, the frame closes.
8	`return n * fact(n - 1);`	3	main | fact(3)	*3 · 2 = 6* is returned, the frame closes.
9	`printf(…, fact(3))`	—	main	6 is printed.

Table 3.1: Trace of `fact(3)` and the call stack.

The first half of the trace is a descent. At step 3 `fact(3)` starts computing the expression `3 * fact(2)` but cannot do the multiplication, because the second factor is not yet known. So it calls `fact(2)` and its own frame waits on the stack. The same happens for `fact(2)` at step 5. At step 6 `fact(1)` reaches the base case and returns 1 without making any call. At this moment there are four frames on the stack: `main` and three calls of `fact`.

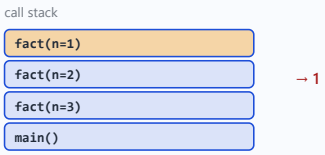


Figure 3.2: The call stack when `fact(1)` reaches the base case. Each of the three `fact` frames has a separate `n`.

The second half is an ascent. The frame of `fact(1)` is removed and control returns to the place where `fact(2)` left off, the unfinished multiplication. In the frame of `fact(2)` the `n` is still 2. That is why it matters that each box in Figure 3.2 has its own copy of `n`. While `fact(1)` was running, `n`

was 1, but this was only in the frame of `fact(1)`. The multiplication gives $2 \cdot 1 = 2$ and `fact(2)` ends. In the same way `fact(3)` computes $3 \cdot 2 = 6$. Finally `main` prints 6 and only `main` is left on the stack.

This trace shows the general behavior of recursive functions. The calls first descend to the base case; the real computation is mostly done on the way back. In the factorial all the multiplications are done during the ascent, from the innermost call outward: first $2 \cdot 1$, then $3 \cdot 2$.

3.2.3 The memory cost of recursion

Every recursive call means a new frame. Frames go away only when the call ends. Consequently the number of frames on the stack at any moment equals the current length of the recursion chain. The largest value of this length is called the **recursion depth**.

Theorem 3.1 (Stack cost of recursion)

For a function with recursion depth d and frames of constant size, the extra memory used on the call stack is $O(d)$.

Proof. The size of a frame depends on the number of parameters and local variables; these numbers are fixed by the code and do not change with the input. Let the frame size be c bytes. At any moment there are at most d recursive frames on the stack, because the recursion depth is the greatest length the call chain can reach. The memory used is at most $c \cdot d$ bytes, that is, $O(d)$. □

For `fact(n)` the depth is n: the chain `fact(n)`, `fact(n - 1)`, ..., `fact(1)` opens n frames. The extra memory is $O(n)$. The time cost is also $\Theta(n)$, because each of the n calls does a constant number of operations (one comparison, one multiplication, one return).

The size of the call stack is limited. The operating system sets aside a fixed stack area for each thread; this area is 1 MB by default on Windows and mostly 8 MB on Linux. Even a function that spends a few tens of bytes per frame fills this area if it descends to hundreds of thousands of levels, and the program crashes with a stack overflow. Recursion of depth n is therefore dangerous for large n; recursion of depth $\log n$, on the other hand, causes no trouble in practice. Even binary search on an array of a billion elements descends no deeper than 30 frames.

Common mistake

Computing only the time cost of a recursive function and forgetting the stack cost. `fact` and the factorial with a loop run in the same $\Theta(n)$ time, but the recursive version uses $O(n)$ and the loop version $O(1)$ extra memory. The stack cost of a recursive algorithm is the total size of the frames in the deepest call chain.

3.3 Recursion or a loop?

It is often possible to solve the same problem both with recursion and with a loop. Which to choose depends on a balance among readability, memory, and speed.

3.3.1 Two factorials

The two versions of the factorial side by side:

```
/* recursive */
long fact(int n) {
  return n <= 1 ? 1 : n * fact(n - 1);
}

/* with a loop */
long fact_loop(int n) {
  long f = 1;
  while (n > 1)
    f *= n--;
  return f;
}
```

The recursive version has shrunk to a single line. The conditional operator `?:` is the expression form of the `if-else` construct: if the condition is true it gives the value before the colon, otherwise the one after. The loop version uses an accumulating variable named `f`. The line `f *= n--;` does two things: first it multiplies `f` by `n`, then it decrements `n` by one. The postfix `--` operator uses the old value in the expression and decrements the variable afterwards. In the call `fact_loop(4)`, `f` becomes 4, 12, 24 in turn, and the loop stops when $n = 1$.

In the loop version, changing the parameter `n` does not affect the caller. In C parameters are passed by value: the function works with a copy of the caller's variable. After the call `fact_loop(x)`, `x` remains unchanged.

3.3.2 Comparison

The differences between the two approaches are collected in Table 3.2.

Criterion	Recursive	Loop
Readability	Very close to the definition	Needs state variables
Extra memory	*O(depth)* stack	*O(1)*
Call cost	One frame per step	None
Risk	Stack overflow in deep recursion	Infinite loop

Criterion	Recursive	Loop
Where it is natural	Trees, divide and conquer, backtracking	Flat scans, counters

Table 3.2: Comparison of recursion and a loop.

The "call cost" row of the table is about speed. At every call a frame is set up, parameters are copied, and on return the frame is removed. These jobs are a few machine instructions, but more expensive than one iteration of a loop. On a simple job such as the factorial, the recursive version runs a little slower than the loop version. The asymptotic order is the same; the difference lies only in the constant factor.

The "where it is natural" row is more important. In the factorial a loop is as clear as recursion. In tree traversal, in the towers of Hanoi, or in backtracking this is not so. In these problems a call makes several subcalls and one must remember where each subcall left off. Recursion leaves this bookkeeping to the call stack; the loop version asks the programmer to keep the same record by hand.

3.3.3 Turning recursion into a loop

Every recursive algorithm can be turned into a loop, and every loop into recursion. The method of conversion is to imitate the call stack: the programmer defines an array or stack structure, puts into it the information that recursion would have saved, and processes it with a loop. The following function computes the factorial this way:

```
long fact_stack(int n) {
  int stack[64], top = 0;       /* array imitating the call stack */
  long f = 1;
  while (n > 1)                   /* descent: push the factors */
    stack[top++] = n--;
  while (top > 0)                 /* return: do the pending products */
    f *= stack[--top];
  return f;
}
```

The first loop corresponds to the descent phase of recursion. In each round it pushes a pending factor (n, $n - 1$, ..., 2) into the array. The second loop is the ascent phase: it takes the factors in reverse order and multiplies them into the accumulator. The size of the array `stack` is chosen as 64; since the factorial no longer fits a 64-bit number at $n = 21$, this size is sufficient. The information the recursive `fact` keeps on the call stack is exactly this array. For the factorial this conversion is an unnecessary trouble, because the multiplications are independent of order and a single variable suffices. In problems such as tree traversal, however, an explicit stack is the most natural way to a nonrecursive solution.

The choice is a balance. If code readability comes first and the depth is small (for example *O(log n)*), recursion is a good choice. If the depth can be large or the cost of each call matters, a loop is preferred.

3.4 Tail recursion

Some recursive functions have a special form: the recursive call is the last thing the function does. Functions of this form can be converted into loops mechanically, and the compiler can do this conversion itself.

3.4.1 Definition and the accumulating parameter

> **Definition 3.4**
>
> If the recursive call is the last job of the function, that is, no work remains to be done after the call returns, this recursion is called **tail recursion**. Such a call is called a **tail call**.

`fact` is not tail recursive. In the line `return n * fact(n - 1);` a multiplication is done after the recursive call returns. The value of `n` must be saved for this multiplication; so the frame has to stay on the stack until the subcall ends. The tail-recursive version of the factorial uses an extra parameter to get rid of the pending multiplication:

```
long fact_tail(int n, long acc) {
  if (n <= 1) return acc;
  return fact_tail(n - 1, n * acc);  /* last job: the call */
}
/* first call: fact_tail(n, 1) */
```

`acc` (accumulator) is the **accumulating parameter** and carries the partial product computed so far. The multiplication is done *before* the recursive call, while the argument is computed: `n * acc`. When the call returns, no work remains; the returned value is passed upward as is. In the base case the accumulator is the full result and is returned directly. In the first call the accumulator is 1, because 1 is the identity of multiplication.

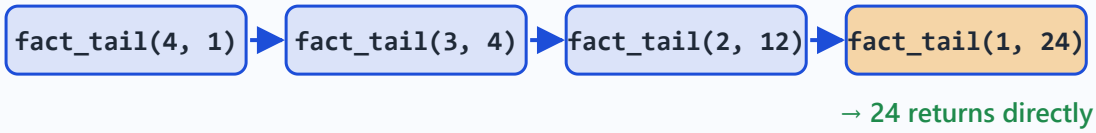


Figure 3.3: The call chain of `fact_tail(4, 1)`. The partial product advances in the parameter; the 24 returned by the innermost call is passed upward without any further operation.

Example 3.1

Tracing the call `fact_tail(4, 1)`:

1. `fact_tail(4, 1)`: $n = 4 > 1$; `fact_tail(3, 4 · 1) = fact_tail(3, 4)` is called.
2. `fact_tail(3, 4)`: `fact_tail(2, 3 · 4) = fact_tail(2, 12)` is called.
3. `fact_tail(2, 12)`: `fact_tail(1, 2 · 12) = fact_tail(1, 24)` is called.
4. `fact_tail(1, 24)`: base case; `acc = 24` is returned.
5. 24 passes through three frames without any further operation and reaches the original caller.

In `fact` the multiplications were done on the way back, from the inside out: $2 \cdot 1, 3 \cdot 2, 4 \cdot 6$. In `fact_tail` they are done on the way down, from the outside in: $4 \cdot 1, 3 \cdot 4, 2 \cdot 12$. Because multiplication is commutative and associative, the result is the same.

3.4.2 The compiler and the tail call

Since after a tail call there is no work waiting in the frame, there is no point in keeping the frame. Instead of opening a new frame, the compiler updates the parameters in the existing frame with the new values and **jumps** to the start of the function. This transformation is called tail call optimization. The result is the same machine code as the loop version, and the stack cost drops to $O(1)$.

The C standard does **not guarantee** this optimization. GCC and Clang often turn tail calls into jumps at the `-O2` level; in a build with optimization off (`-O0`), each call opens a new frame. It is therefore not right in C to rely on a tail recursion whose depth could be very large. Such a function is easy to turn into a loop by hand: the parameters become loop variables, new values are assigned to the parameters in place of the recursive call, and the base case becomes the exit condition of the loop. In this way `fact_tail` turns into `fact_loop`.

Common mistake

Taking a call that stands at the end of a line for tail recursion. In the line `return n * fact(n - 1);` the call appears last in the text, but the last operation performed is the multiplication. The criterion is not the order of writing but whether any work remains after the call returns.

In the line `return fact_tail(n - 1, n * acc);` the result of the call is returned as is; this is a tail call.

3.5 Binary search

Searching is the job of finding a desired value in a data set. The best that can be done in an unsorted array is to look at the elements one by one; this is called linear search and needs *n* comparisons in the worst case. If the array is sorted, much better can be done. In a sorted array a single comparison tells in which half the sought value lies. Binary search uses this information at every step.

3.5.1 The problem and the COMPARE macro

Definition 3.5

Binary search is a routine that finds the sought value in an array sorted in ascending order. At each step it looks at the element in the middle of the range. If the middle element equals the sought value, it returns its index. If it is smaller than the sought value, it discards the left half; if larger, the right half, and continues the search in the remaining half. If the range becomes empty, the value is not in the array and the result is *−1*.

All the binary search examples in this chapter use the same test array:

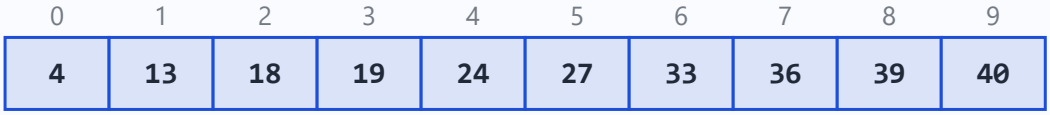


Figure 3.4: The test array `list` : ten elements in ascending order. The numbers above the boxes are indices.

The routine returns not the sought value itself but its **index**. There are two reasons. First, the sought value is already known; what one really wants to know is its place in the array. Once the index is known, the other fields of the record at that position can also be reached. Second, the "not found" information must not be confused with values. Had the routine returned `list[i]`, the *−1* chosen as the error marker could be confused with a real *−1* element in the array. Since array indices in C start from 0, *−1* matches no valid index and safely carries the meaning "absent".

The result of a comparison is one of three cases: less, equal, or greater. To keep the code simple, a macro is defined that turns these three cases into a number:

```
#define COMPARE(x, y) ((x) < (y) ? -1 : (y) < (x) ? 1 : 0)
```

`COMPARE(x, y)` gives *−1* if *x < y*, 1 if *x > y*, and 0 if they are equal. The macro is written with two nested conditional operators. If the first condition `(x) < (y)` is true, the result is *−1*. Otherwise the second condition `(y) < (x)` is tested; if true the result is 1, otherwise 0. Wrapping the macro parameters in parentheses at every occurrence follows from the text substitution rule of the preprocessor. In a call such as `COMPARE(a + 1, b)` the parentheses prevent the expression from being grouped wrongly because of operator precedence.

The result of the macro can be tied to three separate branches in a `switch` statement. For numeric comparison the difference `x - y` could also have been used, but with large numbers the difference can overflow. Since the macro does no subtraction, it does not carry this risk.

3.5.2 Iterative binsearch()

The version of binary search written with a loop is this:

```
int binsearch(int list[], int searchnum,
              int left, int right) {
  while (left <= right) {
    int middle = (left + right) / 2;
    switch (COMPARE(list[middle], searchnum)) {
      case -1: left = middle + 1; break;
      case 0: return middle;
      case 1: right = middle - 1;
    }
  }
  return -1;
}
```

The function takes four parameters. `list` is the starting address of the sorted array, and `searchnum` is the value to be found. `left` and `right` are the left and right end indices of the range in which the search continues. To search the whole array, `binsearch(list, x, 0, n - 1)` is called. The ends of the range are closed: the elements at indices `left` and `right` are part of the range too.

The condition `while (left <= right)` checks that the range is not empty. In a closed range the case *left = right* is a range of one element and must still be searched; that is why the condition is written with `<=`, not `<`. In each round of the loop the middle index is computed with `middle = (left + right) / 2`. In C the quotient of two integers is an integer with the fractional part dropped; *(0 + 9)/2* gives 4.

The `switch` statement divides into three branches according to the result of `COMPARE(list[middle], searchnum)`:

- `case -1`: the middle element is smaller than the sought value. The sought value, if present, is to the right of `middle`. The left end is set to `middle + 1`; `middle` is no longer in the range, because it has been looked at. `break` makes control leave the `switch` and return to the start of the loop.
- `case 0`: equality. The sought value has been found and its index is returned. `return` leaves the `switch`, the loop, and the function all at once.
- `case 1`: the middle element is larger than the sought value. The sought value, if present, is to the left of `middle`; the right end is set to `middle - 1`. Since this is the last branch, the `switch` ends anyway even without a `break`.

The loop ends when the range becomes empty (*left* > *right*) and the function returns *−1*. In every round of the loop the range shrinks by at least one element: either a branch ends with `return`, or `left` increases, or `right` decreases. The loop therefore always terminates.

Common mistake

Updating the ends as `left = middle` or `right = middle`. When two elements remain in the range, for example with *left = 2*, *right = 3*, `middle` becomes 2. If the sought value is on the right, the assignment `left = middle` leaves `left` at 2 and the loop goes around forever with the same range. Adding *±1* both guarantees termination and removes the element already looked at from the range.

3.5.3 Code trace: searching for 18

Tracing the call `binsearch(list, 18, 0, 9)` shows how the range narrows. Table 3.3 gives the variable values in each round.

Round	left	right	middle	list[middle]	COMPARE	Action
1	0	9	4	24	1	`right = 3`
2	0	3	1	13	*−1*	`left = 2`
3	2	3	2	18	0	`return 2`

Table 3.3: Trace of `binsearch(list, 18, 0, 9)`.

In the first round the range is the whole array. The middle index is *(0 + 9)/2 = 4*, and the middle element is 24. Since *24 > 18*, `COMPARE` gives 1: if 18 is present it is to the left of 24. The right half, indices 4–9, is discarded with a single comparison and `right = 3`. In the second round the middle is *(0 + 3)/2 = 1* and the element is 13. Since *13 < 18*, the left side is discarded: `left = 2`. In the third round the middle is *(2 + 3)/2 = 2* and `list[2] = 18` equals the sought value. The function returns 2. In an array of ten elements the value has been found with three comparisons.

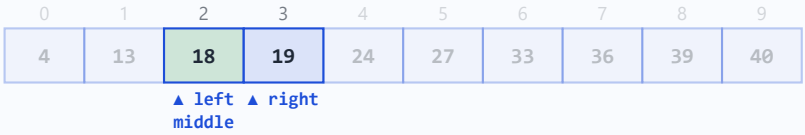


Figure 3.5: The third round: `left = middle = 2`, `right = 3`. The gray cells are the discarded parts; `list[2] = 18` is found.

3.5.4 Code trace: 25 is not in the array

An unsuccessful search shows how the loop ends. The call `binsearch(list, 25, 0, 9)` proceeds as in Table 3.4.

Round	`left`	`right`	`middle`	`list[middle]`	`COMPARE`	Action
1	0	9	4	24	*−1*	`left = 5`
2	5	9	7	36	1	`right = 6`
3	5	6	5	27	1	`right = 4`
—	5	4	—	—	—	*left* > *right*: `return -1`

Table 3.4: Trace of `binsearch(list, 25, 0, 9)`.

In the first round *24 < 25* comes out and the search moves to the right half: `left = 5`. In the second round the middle is *(5 + 9)/2 = 7*; since *36 > 25*, `right = 6`. In the third round the middle is *(5 + 6)/2 = 5* and the element is 27; since *27 > 25*, `right = 4`. Now *left = 5 > right = 4* and the range is empty. The loop condition comes out false and the function returns *−1*.

When the loop ends, the place `left` points to also carries information. *left = 5* is the index where 25 would be placed without breaking the order if it were in the array: between 24 and 27. Throughout the loop every element to the left of `left` stays smaller than the sought value and every element to the right of `right` stays larger. When the range empties, these two regions meet and `left` points to the first larger element. Programs that insert into a sorted array use this observation.

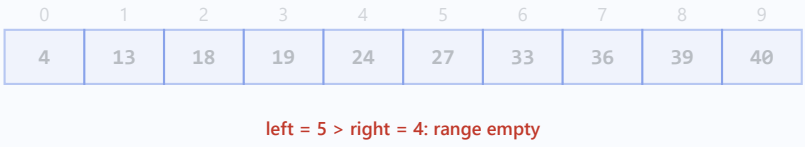


Figure 3.6: The end of the unsuccessful search: `left = 5 > right = 4`, the range is empty and the function returns *−1*.

3.5.5 Recursive binsearch()

Binary search has a recursive structure: a search in half of the range is also a binary search. The version that, instead of a loop, calls itself with a shrunken range is this:

```
int binsearch(int list[], int x, int l, int r) {
  int m;
  if (l > r) return -1;          /* range empty */
  m = (l + r) / 2;
  switch (COMPARE(list[m], x)) {
    case -1: return binsearch(list, x, m + 1, r);
    case 0:  return m;
    case 1:  return binsearch(list, x, l, m - 1);
  }
  return -1;                     /* unreachable */
}
```

To keep the code short, the parameter names are abbreviated: `x` is the sought value, `l` and `r` the left and right ends of the range, and `m` the middle index. The function has two base cases. The first is the range being empty ($l > r$); the value has not been found and -1 is returned. The second is the `case 0` branch; the value has been found and its index is returned. The other two branches are the recursive step. If the middle element is smaller, the search continues in the range $[m + 1, r]$, and if larger, in the range $[l, m - 1]$.

Both recursive calls begin with `return`: the result of the subcall is passed upward unchanged. These calls are tail calls. The line `return -1;` at the end of the function is never reached, because `COMPARE` gives only -1, 0, or 1, and all three branches end with `return`. The reason for writing the line is the compiler: a compiler that cannot know that the `switch` covers every possibility sees a path that ends without returning a value and issues a warning. This line prevents that warning.

Common mistake

Writing `while` instead of `if` in the recursive version. In a body that begins with `while (l <= r)`, every branch leaves with `return`, so the loop runs only once and the result comes out right. Still the code is misleading: a reader expects a loop, whereas the repetition is provided by recursion. In the recursive version the range check must be done with `if`.

Common mistake

Not returning the result of the recursive call. If `case -1: binsearch(list, x, m + 1, r); break;` is written, the subcall finds the value correctly but the returned value is thrown away. The function then reaches the line `return -1;` and returns -1 even when the sought value is in the array. The result of a recursive call must either be returned or used in a variable.

3.5.6 Code trace: the recursive version and the call stack

The recursive call `binsearch(list, 18, 0, 9)` tries the same middle indices (4, 1, 2) as the iterative version. The difference is that each round is done in a separate frame. Table 3.5 shows the changes in the call stack.

Step	Stack (bottom to top)	`l, r, m`	Event
1	main | bs(0, 9)	0, 9, 4	`list[4] = 24`
2	main | bs(0, 9)	0, 9, 4	*24 > 18*, `case 1`: `binsearch(list, 18, 0, 3)`
3	main | bs(0, 9) | bs(0, 3)	0, 3, 1	`list[1] = 13`
4	main | bs(0, 9) | bs(0, 3)	0, 3, 1	*13 < 18*, `case -1`: `binsearch(list, 18, 2, 3)`
5	main | bs(0, 9) | bs(0, 3) | bs(2, 3)	2, 3, 2	`list[2] = 18`
6	main | bs(0, 9) | bs(0, 3) | bs(2, 3)	2, 3, 2	`case 0`: `return 2`
7	main | bs(0, 9) | bs(0, 3)	0, 3, 1	2 returns unchanged from the `case -1` frame
8	main | bs(0, 9)	0, 9, 4	2 returns unchanged from the `case 1` frame
9	main	—	2 is returned to `main`

Table 3.5: Trace of the recursive `binsearch(list, 18, 0, 9)`.

In the table `bs(l, r)` abbreviates the call `binsearch(list, 18, l, r)`. In the descent phase each call halves the range and opens a new frame. At the deepest point, in steps 5 and 6, there are three `binsearch` frames on the stack above `main`. When the base case returns 2, the ascent begins. `bs(2, 3)` returns 2 to `bs(0, 3)`, which is waiting in its `case -1` branch. That in turn returns 2, without doing any operation, to `bs(0, 9)`, which is waiting in its `case 1` branch. Finally 2 reaches `main`.

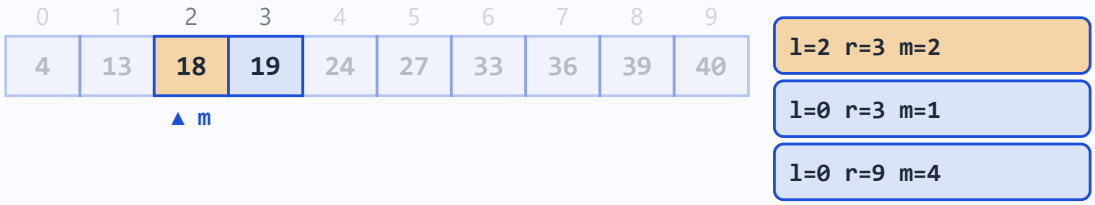


Figure 3.7: The deepest moment of the recursive binary search. The three frames on the right hold the ranges *[0, 9]*, *[0, 3]*, and *[2, 3]*; the topmost frame finds the value at *m = 2*.

In the ascent phase no computation is done. In the factorial there were multiplications on the way back; here the returned value is passed on unchanged. This difference comes from whether the two functions are tail recursive. If tail call optimization is on, the compiler reduces these three frames to a single frame and the code works the same way as the iterative version.

In an unsuccessful search the depth can be one more. The call `binsearch(list, 25, 0, 9)` opens four frames with the ranges *[0, 9]*, *[5, 9]*, *[5, 6]*, and *[5, 4]*. The last frame sees the empty range and immediately returns *−1*. In the iterative version this last check corresponds to the loop condition coming out false.

3.5.7 Complexity

The speed of binary search comes from the range shrinking by at least half in every round. The number of elements in the range shrinks from n to $n/2$, $n/4$, ..., and the last comparison is made when it has dropped to 1 at the latest.

Theorem 3.2 (Round bound of binary search)

In a sorted array of $n \geq 1$ elements, binary search, whether the value is found or not, runs at most $\lfloor \log_2 n \rfloor + 1$ rounds. The time cost is $O(\log n)$.

Proof. Let there be s elements in the range at the start of a round. After the middle element is removed, $\lfloor (s - 1)/2 \rfloor$ elements remain on the left and $\lceil (s - 1)/2 \rceil = \lfloor s/2 \rfloor$ on the right. Whichever half is taken, the new range has at most $\lfloor s/2 \rfloor$ elements. Since initially $s = n$, after k rounds at most $\lfloor n/2^k \rfloor$ elements remain in the range. For round $(k + 1)$ to be possible, the range must not be empty: $n/2^k \geq 1$, that is, $k \leq \log_2 n$. The round number $k + 1$ can be at most $\lfloor \log_2 n \rfloor + 1$. Since a constant number of operations is done in each round, the time is $O(\log n)$. □

The same result is obtained with a recurrence relation. Let $T(n)$ be the cost of searching in a range of n elements. One round halves the range at a constant cost c:

$$T(n) = T(n/2) + c,\ T(1) = c$$

Expanding the relation gives $T(n) = T(n/4) + 2c = T(n/8) + 3c = \ldots = T(n/2^k) + kc$. When $n/2^k = 1$, that is, for $k = \log_2 n$, one gets $T(n) = c(\log_2 n + 1)$. This is $O(\log n)$.

Table 3.6 shows what logarithmic growth means. Linear search looks at every element of the array in the worst case; binary search looks at only a few.

n	Binary search (max rounds)	Linear search
10	4	10
1,000	10	1,000
1,000,000	20	1,000,000
1,000,000,000	30	1,000,000,000

Table 3.6: Numbers of rounds (comparisons) in the worst case.

In the test array $n = 10$ and at most $\lfloor \log_2 10 \rfloor + 1 = 3 + 1 = 4$ rounds are needed. For 18 three rounds sufficed. An example that needs four rounds is 40: the middle indices 4, 7, 8, and 9 are tried in turn. These numbers were verified by searching for all values from 0 to 41 in the test array; no search exceeds four rounds. On an array of a million elements, too, when all successful and unsuccessful searches were tried, the maximum number of rounds came out as 20.

The memory cost differs in the two versions. The iterative version uses only a few variables; the extra memory is *O(1)*. The recursive version opens a frame for each round; the depth equals the number of rounds and the extra memory is *O(log n)*. For a billion elements that means 30 frames; in practice it is negligible.

Binary search has two preconditions. First, the array must be sorted; in an unsorted array the decision to discard half would be wrong and an existing value might not be found. Second, the element at any index must be reachable in *O(1)* time. An array satisfies this condition: `list[middle]` is found in one step by adding `middle` element sizes to the starting address. In a linked list, however, reaching the middle node requires moving node by node from the beginning. Each round takes *O(n)* steps and the gain of binary search is lost.

Common mistake

Computing the middle index with `(left + right) / 2` on very large arrays. With a 32-bit `int`, the sum of two indices can exceed $2^{31} - 1$; this is signed overflow and undefined behavior. This happens only in arrays of more than a billion elements, but when it does, it is a hard bug to find. The safe form is `left + (right - left) / 2`; the difference never overflows and the result is the same.

3.6 The towers of Hanoi

The factorial and binary search make a single recursive call at each step. The towers of Hanoi is the first example that makes two recursive calls. Two calls make the number of calls grow quickly and require a new tool to compute the cost of the algorithm: the recurrence relation.

3.6.1 The problem

There are three towers (pegs): source, auxiliary, and destination. On the source tower sit *n* disks. The disks have different sizes and each disk is larger than the one on top of it; the largest disk is at the bottom and the smallest on top. The goal is to move all the disks to the destination tower in the same order, with the help of the auxiliary tower. There are two rules:

1. In each move only one disk is moved; the topmost disk of a tower is taken and put on top of another tower.
2. A disk cannot be placed on top of a smaller disk.

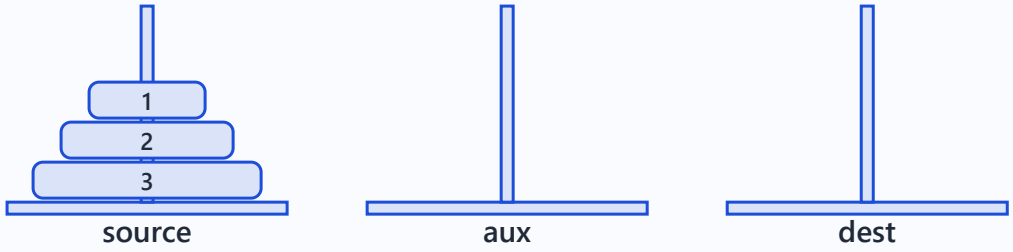


Figure 3.8: The towers of Hanoi with three disks, initial state. The disks are on the source tower, arranged from largest to smallest.

With one or two disks the problem is easy. A single disk is moved directly to the destination. With two disks, the small disk is moved to the auxiliary tower, the large disk to the destination, and the small disk to the destination: three moves. With three disks a solution can be found by trial and error, but with ten disks this way does not work. A systematic method independent of the number of disks is needed.

3.6.2 The recursive solution

The key to the solution is to focus on the largest disk. At some point the largest disk must be moved to the destination tower. At that moment there can be no disk on top of the largest disk, because only the topmost disk can be moved. The destination tower must also be empty, because the largest disk cannot be placed on any disk. All the remaining $n - 1$ disks must then be on the auxiliary tower. This observation divides the solution into three steps:

1. The top $n - 1$ disks are moved from the source to the auxiliary tower, with the help of the destination tower.
2. The largest disk is moved from the source to the destination.
3. The $n - 1$ disks are moved from the auxiliary to the destination tower, with the help of the source tower.

The first and third steps are an instance of the same problem with $n - 1$ disks; only the roles of the towers have changed. In these steps the largest disk sits at the bottom of either the source or the destination. In both cases, because it is larger than all the other disks, it blocks no move; one can act as if it were not there. The base case is a single disk and is solved with one move. The C code of the algorithm is this:

```c
void hanoi(int n, char *src, char *aux,
           char *dst) {
  if (n == 1)
    printf("%s -> %s\n", src, dst);
  else {
    hanoi(n - 1, src, dst, aux);
    printf("%s -> %s\n", src, dst);
    hanoi(n - 1, aux, src, dst);
  }
}
/* main: hanoi(3, "A", "B", "C"); */
```

The towers are represented by their names (strings of type `char *`). The code does not really move disks in memory; it only writes the moves to be made. `printf("%s -> %s\n", src, dst)` reports that the topmost disk of the source tower is moved to the destination tower. If $n = 1$, a single move is written. Otherwise the first recursive call `hanoi(n - 1, src, dst, aux)` is made. In this call one must note that the second and fourth arguments have swapped places: the destination of the $n - 1$ disks is the auxiliary tower of the original problem, and the helper tower is the original destination. Then the move of the largest disk is written. The second call `hanoi(n - 1, aux, src, dst)` moves the $n - 1$ disks from the auxiliary tower to the destination, using the source as the helper.

3.6.3 Code trace: hanoi(3, "A", "B", "C")

In the problem with three disks, A is the source, B the auxiliary, and C the destination. The order of the calls and the moves written are given in Table 3.7. The calls in the table have the form `hanoi(n, src, aux, dst)`.

Step	Active call	Stack depth	Event	Printed
1	`hanoi(3, A, B, C)`	1	to move 2 disks A → B, `hanoi(2, A, C, B)` is called	
2	`hanoi(2, A, C, B)`	2	to move 1 disk A → C, `hanoi(1, A, B, C)` is called	
3	`hanoi(1, A, B, C)`	3	base case	`A -> C`
4	`hanoi(2, A, C, B)`	2	disk 2 is moved; then `hanoi(1, C, A, B)`	`A -> B`
5	`hanoi(1, C, A, B)`	3	base case	`C -> B`
6	`hanoi(3, A, B, C)`	1	disk 3 is moved; then `hanoi(2, B, A, C)`	`A -> C`
7	`hanoi(2, B, A, C)`	2	to move 1 disk B → A, `hanoi(1, B, C, A)`	
8	`hanoi(1, B, C, A)`	3	base case	`B -> A`
9	`hanoi(2, B, A, C)`	2	disk 2 is moved; then `hanoi(1, A, B, C)`	`B -> C`
10	`hanoi(1, A, B, C)`	3	base case	`A -> C`

Table 3.7: Trace of `hanoi(3, "A", "B", "C")`.

The stack depth is given without counting `main`. The program prints seven lines: `A -> C`, `A -> B`, `C -> B`, `A -> C`, `B -> A`, `B -> C`, `A -> C`. The first three moves carry the two small disks to B. The fourth move takes the largest disk to C. The last three moves carry the two disks on B to C, on top of the largest disk. Every move obeys the rules; this can also be verified by playing out the program's output step by step on a board.

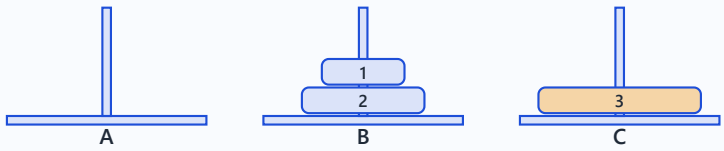


Figure 3.9: The state after the fourth move (step 6): disks 1 and 2 are on the auxiliary tower (B), and the largest disk 3 is on the destination tower (C).

Two numbers in the table stand out. The total number of calls is seven, and each call performs exactly one `printf`; the base cases write one move each, and the other calls write the move in the middle. The stack depth never exceeds 3. In a problem with n disks one call chain descends as $n, n-1, \ldots, 1$; the depth is n and the stack cost is $O(n)$.

3.6.4 The call tree

All the calls of a recursive algorithm can be drawn as a tree. Each node is a call; the subcalls a call makes are its children. In Hanoi every call has two children: the left child is the first recursive call and the right child the second. Figure 3.10 gives the tree of the call `hanoi(3, "1", "2", "3")`, with the towers named 1, 2, and 3. In each box are written, in order, `n`, the source, the auxiliary, and the destination.

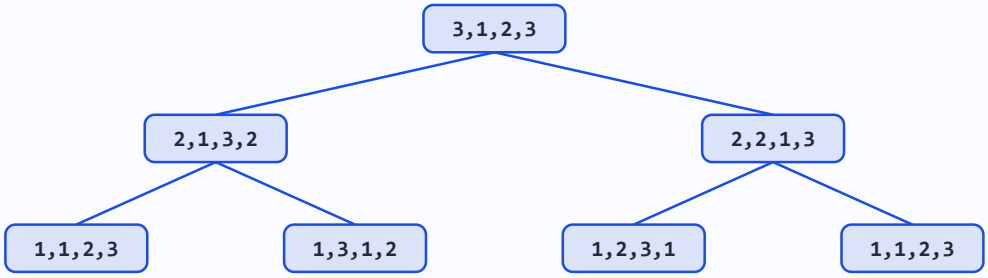


Figure 3.10: The call tree of `hanoi(3, "1", "2", "3")`. Each box is the frame of a call: `n, src, aux, dst`.

The tree carries the whole history of the call stack. At any moment the contents of the stack are the path from the root to the node currently running. For example, while the leftmost leaf (1, 1, 2, 3) is running, the stack contains the root (3, 1, 2, 3), its left child (2, 1, 3, 2), and the leaf itself. The length of the path is at most the height of the tree; this once more shows that the stack depth is n.

The order of the moves is obtained by an **inorder** traversal of the tree: first the left subtree, then the node itself, then the right subtree. The order in the code is the same: the first recursive call, the `printf`, the second recursive call. The move of each node is "source → destination" and is written after its left subtree has finished. This traversal gives the following seven moves:

> 1→3 · 1→2 · 3→2 · 1→3 · 2→1 · 2→3 · 1→3

When the program is run with the same call, this order also comes out.

3.6.5 The recurrence: how many moves?

Let $h(n)$ be the number of moves of the problem with n disks. The algorithm moves $n - 1$ disks twice and makes one move in between. The base case is a single move:

> $h(1) = 1,\ h(n) = h(n-1) + 1 + h(n-1) = 2 \cdot h(n-1) + 1$

> **Definition 3.6**
>
> A **recurrence relation** is an equation that expresses the value of a function in terms of the values of the same function at smaller inputs. To **solve** a recurrence is to write the function as a closed formula that does not refer to itself.

The cost of recursive algorithms is naturally expressed with a recurrence relation, because the algorithm itself has the same form. The simplest solution method is to expand the relation by substituting repeatedly:

```
h(1) = 1
h(n) = 2·h(n-1) + 1
     = 4·h(n-2) + 2 + 1
     = 8·h(n-3) + 4 + 2 + 1
     = …
     = 2ⁿ⁻¹·h(1) + 2ⁿ⁻² + … + 1
     = 2ⁿ - 1
```

In each line $h(n-k)$ is replaced by $2 \cdot h(n-k-1) + 1$. After k steps the coefficient becomes 2^k and the sum $2^{k-1} + \ldots + 2 + 1$ accumulates on the right. At $k = n - 1$ the base case is reached: $h(n) = 2^{n-1} \cdot h(1) + 2^{n-2} + \ldots + 2 + 1$. Since $h(1) = 1$, this is the geometric sum $2^{n-1} + 2^{n-2} + \ldots + 1$ and equals $2^n - 1$. The value of the sum is easily seen with binary numbers: the number made of n one-bits is one less than 2^n.

Theorem 3.3 (Number of Hanoi moves)

The recursive algorithm moves n disks in $2^n - 1$ moves. No method that obeys the rules can finish in fewer moves.

Proof. *Number of moves.* Induction: for $n = 1$, $h(1) = 1 = 2^1 - 1$. If $h(n-1) = 2^{n-1} - 1$ is assumed, then $h(n) = 2(2^{n-1} - 1) + 1 = 2^n - 1$.

Lower bound. Let $m(n)$ be the least number of moves needed to move n disks; $m(1) = 1$. The largest disk must be moved at least once. Just before its first move, all the other $n - 1$ disks must have been gathered on the third tower; this takes at least $m(n-1)$ moves. After the last move of the largest disk, those $n - 1$ disks must be brought to the destination, on top of it; this also takes at least $m(n-1)$ moves. Hence $m(n) \geq 2 \cdot m(n-1) + 1$, and by induction $m(n) \geq 2^n - 1$. Since the algorithm makes exactly this number of moves, it is optimal. □

n	Moves
1	1
2	3
3	7
4	15
10	1,023
20	1,048,575
64	18,446,744,073,709,551,615

Table 3.8: Number of moves $2^n - 1$ by number of disks.

Table 3.8 shows the speed of exponential growth. Each new disk doubles the number of moves. According to legend, priests in a temple are moving a tower of 64 disks, and when the job is finished the world will end. $2^{64} - 1 \approx 1.8 \cdot 10^{19}$ moves, at one move per second, take about 585 billion years. This is more than forty times the present age of the universe.

The time cost of the algorithm is $\Theta(2^n)$. The number of calls also equals the number of moves, because each call writes exactly one move, and it satisfies the relation $c(n) = 2c(n-1) + 1$. The stack cost is $O(n)$. By the theorem there is nothing better: writing a cleverer program cannot reduce the number of moves, because this number comes from the problem itself. Hanoi is a problem whose output length is exponential, and no algorithm can finish without writing the output.

It is sometimes said that some problems can be solved only with recursion. The theoretical counterpart of this saying is not exactly so: every recursion can be turned into a loop with an explicit stack. What the saying really stresses is that for problems like Hanoi recursion is the only natural expression. Hanoi also has solutions with loops, but none is as clear as the three-line idea above.

3.7 Permutation generation

In Hanoi the output was of exponential length. In permutation generation the output grows even faster: n elements have $n!$ arrangements. This problem uses recursion for the first time inside a loop and introduces the basic idea of backtracking, undoing a choice that was made.

3.7.1 The problem and the recursive idea

> **Definition 3.7**
>
> An arrangement of the elements of a set in a certain order is called a **permutation**. A set of $n \geq 1$ elements has $n! = n \cdot (n-1) \cdot \ldots \cdot 1$ different permutations. The permutation problem is to print all of these arrangements.

The number $n!$ comes from a simple counting rule. Any one of the n elements can come in the first position. One of the remaining $n-1$ elements comes in the second, one of the remaining $n-2$ in the third. The choices are multiplied: $n \cdot (n-1) \cdot \ldots \cdot 1$. For example the set {a, b, c} has $3! = 6$ arrangements: abc, acb, bac, bca, cab, cba.

The same counting rule gives a recursive algorithm. The arrangements of the set {a, b, c, d} divide into four groups:

- those that begin with a and are followed by an arrangement of {b, c, d},
- those that begin with b and are followed by an arrangement of {a, c, d},
- those that begin with c and are followed by an arrangement of {a, b, d},
- those that begin with d and are followed by an arrangement of {a, b, c}.

Each group asks for the arrangements of a set of three elements. The problem with three elements reduces in the same way to problems with two elements, and those to problems with one element. A single element has a single arrangement; this is the base case.

3.7.2 Implementing with swaps: perm()

To turn the idea into code, there is no need to keep the set of "remaining elements" in a separate array. All the elements sit in a single character array. The index `i` shows which position of the array is currently being filled. The positions `list[0..i-1]` are the fixed prefix; in the positions `list[i..n-1]` are the elements not yet placed. Each candidate is brought to position `i` in turn **by a swap**, the remaining positions are arranged by recursion, and then the array is restored to its former state **by swapping back**.

A macro with three arguments is used for the swap:

```
#define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
```

The macro exchanges the values of `x` and `y` through the temporary variable `t`. The three assignments are combined into a single expression with the comma operator; the comma operator evaluates expressions from left to right in order. The reason the macro is preferred to a function is that in C a function must take addresses (pointers) in order to change the caller's variables. The macro, on the other hand, is substituted as text in the preprocessor and works directly on `list[i]` and `list[j]`. The permutation function is this:

```
void perm(char *list, int i, int n) {
  int j, temp;
  if (i == n) {                  /* one arrangement */
    for (j = 0; j < n; j++) printf("%c", list[j]);
    printf("\n");
  } else
    for (j = i; j < n; j++) {
      SWAP(list[i], list[j], temp);
      perm(list, i + 1, n);
      SWAP(list[i], list[j], temp);
    }
}
```

`perm(list, i, n)` prints all arrangements of the remaining positions while the first `i` positions of the array `list` are fixed. `i == n` is the base case: all positions are fixed and the array is a permutation. In this case the first `for` loop prints `n` characters, followed by a line break. If it is not the base case, the second `for` loop takes `j` from `i` to `n - 1`. Three things are done in each round:

1. `SWAP(list[i], list[j], temp)`: the element at position `j` is brought to position `i`. When *j = i* the element is swapped with itself and the array does not change.
2. `perm(list, i + 1, n)`: position `i` is now fixed; all arrangements of the remaining positions are printed.
3. `SWAP(list[i], list[j], temp)`: the same swap is repeated and the array returns to its state at the start of the loop round.

The variable `temp` is declared as `int`; a `char` is put into it and assigned back to a `char`. This conversion loses no value, because every `char` value fits in an `int`. The first call of the function, to arrange the whole array, has the form `perm(list, 0, n)`. `list` must be a modifiable array: the declaration `char s[] = "abc";` creates such an array, but giving a string constant, as in `perm("abc", 0, 3)`, is undefined behavior. String constants may live in read-only memory, and the swap can crash the program.

3.7.3 Code trace: perm("abc", 0, 3)

The call `perm(list, 0, 3)` with the array `char list[] = "abc"` prints six arrangements. Table 3.9 gives all the steps of the trace. In the stack column `p(i)` abbreviates the call `perm(list, i, 3)`.

Step	Stack	`i, j`	Action	`list`	Printed
1	p(0)	0, 0	swap with itself, `perm(i = 1)`	abc	
2	p(0) p(1)	1, 1	swap with itself, `perm(i = 2)`	abc	
3	p(0) p(1) p(2) p(3)	3, —	base case	abc	abc
4	p(0) p(1)	1, 2	`list[1] ⇔ list[2]`, `perm(i = 2)`	acb	
5	p(0) p(1) p(2) p(3)	3, —	base case	acb	acb
6	p(0) p(1)	1, 2	swap back `list[1] ⇔ list[2]`	abc	
7	p(0)	0, 1	`list[0] ⇔ list[1]`, `perm(i = 1)`	bac	
8	p(0) p(1)	1, 1	swap with itself, `perm(i = 2)`	bac	
9	p(0) p(1) p(2) p(3)	3, —	base case	bac	bac
10	p(0) p(1)	1, 2	`list[1] ⇔ list[2]`, `perm(i = 2)`	bca	
11	p(0) p(1) p(2) p(3)	3, —	base case	bca	bca
12	p(0) p(1)	1, 2	swap back `list[1] ⇔ list[2]`	bac	
13	p(0)	0, 1	swap back `list[0] ⇔ list[1]`	abc	
14	p(0)	0, 2	`list[0] ⇔ list[2]`, `perm(i = 1)`	cba	
15	p(0) p(1)	1, 1	swap with itself, `perm(i = 2)`	cba	
16	p(0) p(1) p(2) p(3)	3, —	base case	cba	cba
17	p(0) p(1)	1, 2	`list[1] ⇔ list[2]`, `perm(i = 2)`	cab	
18	p(0) p(1) p(2) p(3)	3, —	base case	cab	cab
19	p(0) p(1)	1, 2	swap back `list[1] ⇔ list[2]`	cba	
20	p(0)	0, 2	swap back `list[0] ⇔ list[2]`	abc	

Table 3.9: Trace of `perm("abc", 0, 3)`.

In the table the own loop of the call `p(2)` has been abbreviated. When *i* = 2, `j` takes only the value 2; a swap with itself is made, `p(3)` is called, and the swap back is again with itself. These steps do not change the array.

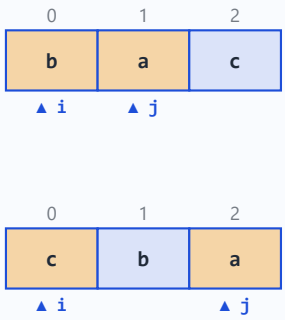


Figure 3.11: Two swaps of the outer loop. (a) *i = 0, j = 1*: a and b change places, the array becomes `bac` (step 7). (b) *i = 0, j = 2*: a and c change places, the array becomes `cba` (step 14).

The output order is abc, acb, bac, bca, cba, cab. The last two lines do not follow dictionary order: cba comes before cab. The reason is the swap at step 14. In the round *j = 2* a and c change places and the array becomes `cba`; b sits in the second position and a in the third. Because the subcall starts in this order, cba is printed first. All the arrangements come out exactly once; only their order is different.

The swaps back at steps 13 and 20 are essential for the correctness of the algorithm. Each round of the outer loop expects to find the array as `abc`. Had the swaps back not been made, the round *j = 1* would have left the array as `cba` and the round *j = 2* would have swapped `list[0]` and `list[2]`, that is, c and a. In that case the arrangements starting with b would never be produced and those starting with a would be produced a second time. When the program is run and the array is printed after the call, `abc` is seen: every call leaves the array as it found it.

Common mistake

Forgetting the second `SWAP`. The code compiles and prints six lines, but some arrangements repeat and some never appear. The error can go unnoticed on small inputs. In every backtracking algorithm, a change made before the recursive call must be undone after the call.

3.7.4 The recursion tree and complexity

The calls of `perm` form a tree. At the root no position is fixed. Each level fixes one more position, and the children of a node are the elements that can come to that position. The leaves are the completed arrangements.

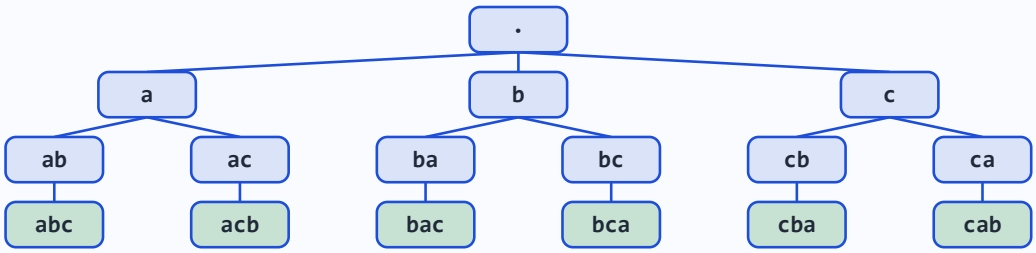


Figure 3.12: The recursion tree of `perm("abc", 0, 3)`. The root has no fixed prefix; each level fixes one position. The leaves give the six arrangements in the order of the output.

In the tree of Figure 3.12 the root has three children (a, b, c), each of them has two children (ab, ac, ...), and those have one child each. Sibling branches must see the same starting array; the second `SWAP` repairs the array for this reason. The number of nodes at each level of the tree gives the number of calls. At level k the first k positions are fixed, and these positions can be filled in $n \cdot (n-1) \cdot \ldots \cdot (n-k+1)$ different ways:

> number of calls = 1 + n + n(n − 1) + ... + n!

For $n = 3$ there are $1 + 3 + 6 + 6 = 16$ calls, for $n = 4$ there are $1 + 4 + 12 + 24 + 24 = 65$. These numbers were also verified by a program, by adding a counter to the function. The last two terms of the sum are $n!$ and the earlier ones shrink quickly; the total is about $e \cdot n! \approx 2.72 \cdot n!$. The number of leaves is exactly $n!$.

What determines the time is the writing at the leaves. Each leaf writes n characters; $n!$ leaves come to $n \cdot n!$ characters in all. The work done at the internal nodes (swaps and loop checks) is constant per call and totals $O(n!)$. The total time is $\Theta(n \cdot n!)$. The permutation problem is sometimes briefly called $O(n!)$ in some sources; this notation describes the number of arrangements but overlooks the factor of n for writing each arrangement. The stack cost equals the depth: $n + 1$ frames, that is, $O(n)$.

Factorial growth is even faster than exponential growth. At $n = 10$ there are $10! = 3{,}628{,}800$ arrangements and the number of characters to be written exceeds 36 million. At $n = 20$ the number of arrangements is $2.4 \cdot 10^{18}$; even a machine that produces a billion arrangements per second finishes this job in 77 years. A solution based on trying all arrangements in turn can therefore be used only for very small n.

3.8 Brute force and backtracking

Permutation and subset generation are ways of systematically walking through all the candidate solutions of a problem. Many problems can be solved by this walk: candidates are generated and each is tested against the rules. The rest of this chapter examines this approach and its clever form, backtracking.

3.8.1 Two approaches

Definition 3.8

In the **brute force** approach all possible candidate solutions are generated and each is tested separately. In the **backtracking** approach the solution is built step by step. The moment a partial solution breaks a rule, all the candidates in that branch are **pruned**: the last choice is undone and the next option is tried.

Brute force relies on the speed of the computer. It is easy to set up and always gives the right answer, because it skips no candidate. Its problem is the number of candidates. A problem made of n yes/no decisions has 2^n candidates, and a problem asking for an ordering of n elements has $n!$. Both reach numbers that no machine could finish once n passes a few dozen.

Backtracking walks through the same candidates, but with one difference: a candidate is tested before it is completed. If a partial solution is seen to break the rule, none of the candidates that begin with that partial solution are generated. For example, when placing eight queens, if the first two queens threaten each other, all placements of the other six queens are eliminated at once. Because this elimination resembles cutting off a branch of a tree, it is called pruning. The worst-case cost often stays in the same order as brute force, but in practice the number of candidates tried becomes much smaller.

One should never be fully sure of the data structure and method chosen when solving a problem. A solution that works by brute force is a starting point that calls for a search for something better. This attitude can be summed up in the sentence "there is a better solution method and we just have not found it". Backtracking and the heuristic rule in the knight's tour are products of such a search.

3.8.2 The general pattern of backtracking

All backtracking algorithms follow the same skeleton. The following pseudocode gives this skeleton:

```
void backtrack(partial) {
  if (is_complete) { print; return; }
  for (each candidate choice)
    if (valid) { choose; backtrack(...); undo; }
}
```

The function takes a partial solution. If the partial solution is complete, it is printed and the function returns; this is the base case. If it is not complete, every candidate choice that could extend the partial solution by one step is tried in turn. A choice that obeys the rules is added to the partial solution (`choose`), the function calls itself, and when the call returns the choice is undone (`undo`). For choices that do not obey the rules, the recursive call is never made; that is pruning.

The permutation code fits this pattern exactly. The partial solution is the fixed prefix. The candidate choices are the elements `list[i..n-1]`. Choosing is the first `SWAP` and undoing the choice is the second `SWAP`. In permutations every choice obeys the rules, so there is no pruning. In the n-queens problem, however, the `valid` test does real work and cuts most of the branches.

Backtracking is sometimes wrongly understood as "rewinding the scenario from the end to the beginning". The method does not build the solution from the end to the start. It builds the partial solution forward; when it hits a dead end, it undoes only the last choice and continues forward from the same point with another choice. The only thing gone back to is the last decision point.

3.8.3 Subset generation

Printing all subsets of a set is the plainest backtracking example. A set of n elements has 2^n subsets, because there are two options for each element: it is taken into the subset or it is not. The algorithm makes these decisions in turn:

```
void subsets(char *s, int n, int i,
             int pick[]) {
  int j;
  if (i == n) {                  /* decisions done */
    printf("{");
    for (j = 0; j < n; j++)
      if (pick[j]) printf("%c", s[j]);
    printf("}\n");
    return;
  }
  pick[i] = 1; subsets(s, n, i + 1, pick);
  pick[i] = 0; subsets(s, n, i + 1, pick);
}
```

`s` is the character array holding the elements, and `n` is the number of elements. The array `pick` holds the decisions: if `pick[j]` is 1, element `j` has been taken, and if 0 it has not. `i` shows for which element the next decision will be made. `i == n` is the base case: all decisions have been made and the elements for which `pick` is 1 are printed inside braces. If it is not the base case, two calls are made. First, with `pick[i] = 1`, element `i` is taken and the function is called for the remaining decisions. Then, with `pick[i] = 0`, the element is left out and the same call is repeated.

In the code the step "undo the choice" does not appear as a separate line. The assignment `pick[i] = 0` does two jobs at once: it undoes the choice of the first branch and makes the choice of the second branch. When leaving the function, `pick[i]` stays 0, but this does no harm. A caller one level up always reassigns `pick[i]` before reading it; and the printing at the base case reads only the values assigned up to that point.

No.	a	b	c	`pick`	Printed
1	take	take	take	[1, 1, 1]	{abc}

No.	a	b	c	`pick`	Printed
2	take	take	skip	[1, 1, 0]	{ab}
3	take	skip	take	[1, 0, 1]	{ac}
4	take	skip	skip	[1, 0, 0]	{a}
5	skip	take	take	[0, 1, 1]	{bc}
6	skip	take	skip	[0, 1, 0]	{b}
7	skip	skip	take	[0, 0, 1]	{c}
8	skip	skip	skip	[0, 0, 0]	{}

Table 3.10: Output of `subsets("abc", 3, 0, pick)` and the decisions.

In Table 3.10 the decisions resemble a binary counter counting backwards: the array `pick` goes 111, 110, 101, …, 000. The reason is that the decision "take" is tried before "skip". The `{}` in the last row is the empty set; the empty set is also a subset of every set.

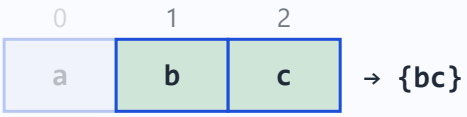


Figure 3.13: The fifth leaf: a is not taken, b and c are taken; `{bc}` is printed.

The complexity computation is similar to Hanoi. Each call makes two calls and the depth is n. The call tree is a perfect binary tree with $n + 1$ levels: it has 2^n leaves and $2^{n+1} - 1$ nodes in all. For $n = 3$ there are 15 calls, for $n = 4$ there are 31. Each leaf does $O(n)$ work by walking through the array `pick` from beginning to end. The total time is $\Theta(n \cdot 2^n)$ and the stack cost is $O(n)$. The calling function allocates the array `pick` once and all calls use the same array; in each frame only a pointer to the array sits.

3.9 The satisfiability (SAT) problem

Subset generation walked over two options for each element. The same walk can also be applied to assigning the values true or false to logical variables. The problem of this section is one of the most famous problems of computer science and shows the limits of brute force clearly.

3.9.1 The problem

Definition 3.9

Let $x_1, x_2, \ldots, x_n$ be logical variables that take the value true (T) or false (F). A formula is built from these variables with the operators ∧ (and), ∨ (or), and ¬ (not). The **satisfiability problem** (SAT for short) asks: is there an assignment of values to the variables that makes the formula true?

The formula examined in this section is:

$$(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_3) \vee \neg x_3$$

The formula is the disjunction of three terms, and if one of the terms is true the formula is true. For example, if $x_3 = F$, the last term $\neg x_3$ is true and so is the formula. Thus one sees at once that the formula is satisfiable. The question is how a computer program will find this.

The brute force solution is to try all assignments. Since each of the three variables takes two values, there are $2^3 = 8$ assignments: {F, F, F}, {F, F, T}, …, {T, T, T}. With n variables the number of assignments is 2^n. In each assignment the formula is evaluated once; if the size of the formula is $|F|$, this takes $O(|F|)$ time. The total cost is $O(2^n \cdot |F|)$.

Exponential cost cannot be overcome by parallel hardware either. Even if the work is divided equally among m processors, the cost is $O(2^n/m)$. In a formula with 100 variables there are $2^{100} \approx 1.3 \cdot 10^{30}$ assignments; even a million processors bring this number down only to the order of 10^{24}.

No polynomial-time algorithm is known for SAT. The problem is the first member of the class of problems called **NP-complete**. If a polynomial-time algorithm is found for one of the problems in this class, all the problems in the class can be solved in polynomial time. So far no such algorithm has been found, but it has not been proved that none exists. The special form in which each clause has at most two variables (2-SAT) is solved in polynomial time; when the clauses have three variables (3-SAT) the problem is NP-complete again. The classification of problems in this way is the subject of the theory of computation course.

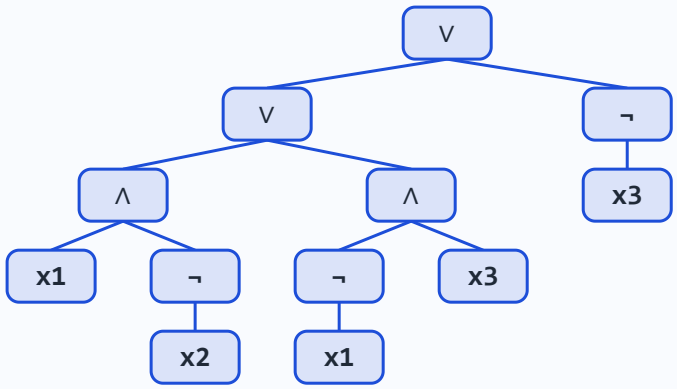


Figure 3.14: The tree of the formula $(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_3) \vee \neg x_3$. Internal nodes are operators, leaves are variables.

3.9.2 The formula tree and postorder evaluation

The natural way to evaluate a formula in a program is to store it as a tree. In the tree of Figure 3.14 each internal node is an operator. The ∧ and ∨ nodes have two children. The ¬ node has a single operand and it sits in the right child. The leaves are variables; the same variable may appear in several leaves. The definition of the tree in C and the evaluation function are as follows:

```
typedef enum { NOT_N, AND_N, OR_N, TRUE_N, FALSE_N } logic;
typedef struct tree_node *tree_ptr;
struct tree_node {
  tree_ptr left, right;
  logic data;
  short value;
};

void post_eval(tree_ptr L) {
  if (L) {
    post_eval(L->left);
    post_eval(L->right);
    switch (L->data) {
      case NOT_N: L->value = !L->right->value; break;
      case AND_N: L->value = L->left->value
                          && L->right->value; break;
      case OR_N:  L->value = L->left->value
                          || L->right->value; break;
      case TRUE_N:  L->value = 1; break;
      case FALSE_N: L->value = 0; break;
    }
  }
}
```

The first line defines an enumeration that names the node types. `NOT_N`, `AND_N`, and `OR_N` are the operator nodes. `TRUE_N` and `FALSE_N` are for the leaves: when an assignment is made, the type of each leaf becomes one of these two, according to the value of the variable in that assignment. The second line defines the name `tree_ptr` to mean "pointer to `struct tree_node`". The structure itself is in the third declaration: two child pointers, the type of the node (`data`), and the computed value of the node (`value`).

Placing the `typedef` before the structure definition is valid in C. At this point `struct tree_node` is still an incomplete type, but to define a pointer to it one need not know the contents of the type. This lets the structure contain fields that point to nodes of its own type. A structure cannot have a field of its own type (the value itself, not a pointer); that would mean a structure of infinite size.

There are two details in the choice of names. English names such as `enum { not, and, or, true, false }` cause trouble. `true` and `false` are defined as macros in the C99 header `<stdbool.h>` and became keywords in C23; `not`, `and`, and `or` are macros standing in for operators in the header `<iso646.h>`. If one of these headers is included, the enumeration does not compile. The second detail is that a `typedef` with no name written at the end, such as `typedef struct tree_node {…};`, defines no name. The compiler accepts this line but issues a warning. The code above uses names that do not clash.

The appearance of the node in memory is also worth examining. On a 64-bit system each of the two pointers takes 8 bytes and they make up the first 16 bytes of the structure. The field `data` (an `enum`, 4 bytes) starts at byte 16 and the field `value` (`short`, 2 bytes) at byte 20. The compiler adds 2 bytes of padding at the end of the structure and rounds the size up to 24 bytes, a multiple of 8. This padding ensures that in an array of structures every pointer sits at an address that is a multiple of 8. These numbers were measured with `sizeof` and `offsetof`; they may differ with another compiler or on a 32-bit system.

The nodes of the tree are created at run time with `malloc`. The code in the slides does not show this part; for a complete program the following two helper functions suffice:

```
tree_ptr new_node(logic v, tree_ptr l, tree_ptr r) {
  tree_ptr p = malloc(sizeof *p);
  if (p == NULL) {              /* allocation failed */
    fprintf(stderr, "out of memory\n");
    exit(EXIT_FAILURE);
  }
  p->data = v;
  p->left = l;
  p->right = r;
  p->value = 0;
  return p;
}

void tree_free(tree_ptr L) {     /* postorder: children first */
  if (L) {
    tree_free(L->left);
    tree_free(L->right);
    free(L);
  }
}
```

`new_node` asks for memory of the size of one node. The expression `sizeof *p` gives the size of the type `p` points to; since no type name is written, the expression stays correct even if the type of `p` changes. If `malloc` cannot allocate memory, it returns `NULL`. If `p->data` were written without checking this, the program would write through a null pointer and crash; that is why the return value is tested at once. `tree_free` gives back all the nodes of the tree. It deletes the two children first, then the node itself. The order matters: had the node been freed first, the field `L->left` would be read from freed memory, which is undefined behavior. If the program ends without the tree being deleted, the operating system reclaims the memory anyway, but in a long-running program, if a new tree is built for each assignment and not deleted, a **memory leak** results.

The formula tree and the assignment are set up like this:

```
tree_ptr leaf[5];   /* leaves: x1, x2, x1, x3, x3 */

tree_ptr build_formula(void) {
  int k;
  for (k = 0; k < 5; k++)
    leaf[k] = new_node(FALSE_N, NULL, NULL);
  /* (x1 AND NOT x2) OR (NOT x1 AND x3) OR NOT x3 */
  return new_node(OR_N,
           new_node(OR_N,
             new_node(AND_N, leaf[0], new_node(NOT_N, NULL, leaf[1])),
             new_node(AND_N, new_node(NOT_N, NULL, leaf[2]), leaf[3])),
           new_node(NOT_N, NULL, leaf[4]));
}

void set_leaves(tree_ptr root, int x1, int x2, int x3) {
  (void)root;            /* leaves are reached directly through leaf[] */
  leaf[0]->data = x1 ? TRUE_N : FALSE_N;
  leaf[1]->data = x2 ? TRUE_N : FALSE_N;
  leaf[2]->data = x1 ? TRUE_N : FALSE_N;
  leaf[3]->data = x3 ? TRUE_N : FALSE_N;
  leaf[4]->data = x3 ? TRUE_N : FALSE_N;
}
```

The five leaves are kept in the array `leaf`. While the tree is built, the addresses of these leaves are given as children to the appropriate operator nodes. `set_leaves` places an assignment into the tree: it makes the type of each leaf `TRUE_N` or `FALSE_N` according to the value of the variable it represents. The structure of the tree does not change; only the leaves change.

`post_eval` computes the value of the tree with a **postorder** traversal: first the left subtree, then the right subtree, and the node itself last. This order is mandatory, because the value of an operator node depends on the values of its children and the children must be computed before it. The function starts with `if (L)`; for an empty subtree (`NULL`) it does nothing. This check stops the recursion both at the leaves and at the empty left child of a ¬ node. The `switch` statement computes the value according to the type of the node. The `NOT_N` node inverts the value of the right child. The `AND_N` and `OR_N` nodes combine the values of the two children with C's `&&` and `||` operators. The leaves take the value 1 or 0 according to their type.

Example 3.2

For the assignment $x_1 = x_2 = x_3 = T$, the tree is evaluated from the bottom up as follows:

1. Leaves: $x_1 = T, x_2 = T, x_1 = T, x_3 = T, x_3 = T$.
2. ¬ nodes: $\neg x_2 = F, \neg x_1 = F, \neg x_3 = F$.
3. ∧ nodes: $x_1 \wedge \neg x_2 = T \wedge F = F, \neg x_1 \wedge x_3 = F \wedge T = F$.
4. The inner ∨: $F \vee F = F$. The ∨ at the root: $F \vee \neg x_3 = F \vee F = F$.

The root comes out false: this assignment does not satisfy the formula.

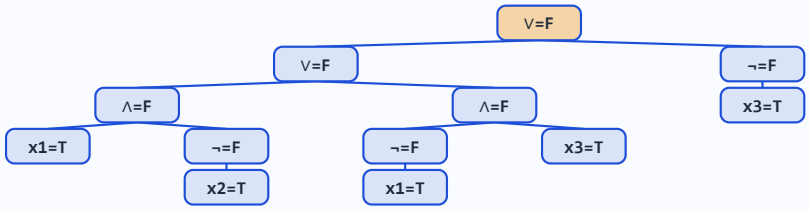


Figure 3.15: The tree evaluated at the assignment 111. No term is true; the root is false.

3.9.3 All assignments by brute force

Three nested loops could be written to walk over the eight assignments. A shorter way is to use a single loop that counts from 0 to 7 and to distribute the bits of the counter among the variables:

```
int m, x1, x2, x3;
for (m = 0; m < 8; m++) {    /* 2^3 assignments */
  x1 = (m >> 2) & 1;
  x2 = (m >> 1) & 1;
  x3 = m & 1;
  set_leaves(root, x1, x2, x3);
  post_eval(root);
  if (root->value)
    printf("%d%d%d\n", x1, x2, x3);
}
```

The three-bit binary representation of `m` is an assignment. The expression `(m >> 2) & 1` shifts `m` two bits to the right and takes the rightmost bit; this is bit number 2 of `m` and is assigned to x_1. In the same way bit 1 goes to x_2 and bit 0 to x_3. For $m = 5$ the binary representation is 101 and one gets $x_1 = 1$, $x_2 = 0$, $x_3 = 1$. In each round the assignment is placed into the tree, the tree is evaluated, and if the root is true the assignment is printed.

m	$x_1x_2x_3$	$x_1 \wedge \neg x_2$	$\neg x_1 \wedge x_3$	$\neg x_3$	Root
0	000	F	F	T	T
1	001	F	T	F	T
2	010	F	F	T	T
3	011	F	T	F	T
4	100	T	F	T	T
5	101	T	F	F	T
6	110	F	F	T	T

m	$x_1x_2x_3$	$x_1 \wedge \neg x_2$	$\neg x_1 \wedge x_3$	$\neg x_3$	Root
7	111	F	F	F	F

Table 3.11: The value of the formula at the eight assignments.

Table 3.11 shows that seven of the eight assignments satisfy the formula. The program prints the lines 000, 001, 010, 011, 100, 101, and 110; only the assignment $x_1 = x_2 = x_3 = 1$ does not satisfy the formula. This output was verified by turning the code into a complete program and running it.

The SAT question asks only "is there an assignment?". To answer it, it is not necessary to print all assignments; a routine that returns at the first assignment that comes out true suffices. Such a routine stops at 000, the first assignment for this formula. Had the formula been unsatisfiable, the routine would have had to try all eight assignments. The worst case of brute force is the case in which the answer is "no".

3.9.4 Kinds of SAT and precomputation

The satisfiability question can be asked in different forms:

Circuit satisfiability (CIRCUIT-SAT)
: Is there an assignment of inputs that makes the output of a given logic circuit 1? A circuit is a network made of gates (AND, OR, NOT).

Formula satisfiability (Formula SAT)
: Whether a propositional formula built with ∧, ∨, and ¬, as in this section, can be made true.

3-CNF-SAT
: The formula is the ∧ of clauses; each clause is the ∨ of three variables (or their negations). Example: $(x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$. CNF stands for conjunctive normal form.

The brute force solution has a notable efficiency detail. The formula tree is built only **once**. At each assignment the structure of the tree stays the same and only the leaves change. Had the tree been built from scratch at each assignment, a chain of `malloc` and `free` would have run eight times. A data structure prepared once for a job that will be done many times is called **precomputing**. Precomputing may not change the order of the algorithm but it greatly reduces the constant factor.

Precomputing is also used in text search. When searching for a word in a text, if the structure of the word itself is examined beforehand and written into a table, there is no need to back up during the search. The Knuth–Morris–Pratt algorithm is a well-known example of this idea. The topics of greedy algorithms, branch and bound, and precomputing, in algorithm analysis books, describe ways of looking for something better than brute force.

3.10 The n-queens problem

In permutation and subset generation every choice was valid; backtracking amounted only to undoing the choice. In the n-queens problem most choices break the rules, and pruning gives a real gain.

3.10.1 The problem and the representation

Definition 3.10

The **n-queens problem**: place n queens on an $n \times n$ chessboard so that none threatens another. In chess a queen threatens every square along its row, its column, and its two diagonals. Hence there cannot be two queens in the same row, in the same column, or on the same diagonal.

Since two queens cannot stand in the same row, the n queens are spread over the n rows one each. This observation simplifies the representation: a solution is an array `q` of n elements, and `q[r]` is the column number of the queen in row r. The row rule is satisfied automatically by this representation. The column rule is that the values in `q` are all different. For the diagonal rule a test is needed.

If two queens at the squares *(i, q[i])* and *(r, c)* are on the same diagonal, the row difference and the column difference are equal. Advancing in columns as much as the distance between rows is walking on a line of 45 degrees. Since the new queen is always placed in a lower row, $r > i$ and the row difference is $r - i$. The column difference can go in two directions; the absolute value is taken:

> |q[i] − c| = r − i

In the brute force solution the queen of each row is put in one of n columns and n^n candidates result. For $n = 8$ this is $8^8 = 16{,}777{,}216$, that is, about 16.8 million candidates. Applying the column rule from the start and trying only the permutations brings the number down to $8! = 40{,}320$. Backtracking tries even fewer.

3.10.2 safe() and place()

The backtracking solution consists of two functions. The first tests whether a square conflicts with the queens in the previous rows:

```
int q[8];  /* q[r]: column of the queen in row r */

int safe(int r, int c) {
  int i;
  for (i = 0; i < r; i++)
    if (q[i] == c || abs(q[i] - c) == r - i)
      return 0;         /* same column or diagonal */
  return 1;
}
```

The array `q` is defined as global, so that both functions reach it without taking it as a parameter. The size of the array is 8, so the code works for *n = 8* at most. `safe(r, c)` says whether a queen can be put in column *c* of row *r*. The loop looks only at rows *0 … r − 1*, because there are no queens in the lower rows yet. For each earlier queen two conditions are tested: `q[i] == c` catches the same column and `abs(q[i] - c) == r - i` catches the same diagonal. If there is a conflict, the function returns 0 at once. If the loop ends without finding a conflict, the square is safe. The function `abs` is in the header `<stdlib.h>`; if this header is not included, the compiler gives a warning or an error for an undeclared function.

The second function places the queens row by row:

```
int place(int r, int n) {
  int c;
  if (r == n) return 1;             /* solution */
  for (c = 0; c < n; c++)
    if (safe(r, c)) {
      q[r] = c;
      if (place(r + 1, n)) return 1;
    }
  return 0;                  /* backtrack */
}
```

`place(r, n)` tries to place the remaining queens starting from row *r*. It returns 1 if it succeeds and 0 if it fails. `r == n` is the base case: all rows have been filled and a solution has been found. If it is not the base case, the loop tries the columns from left to right. When a safe column is found, the queen is put there (`q[r] = c`) and the function calls itself for the next row. If the subcall returns 1, the solution is complete and the 1 is passed upward at once. If the subcall returns 0, this placement has reached a dead end; the loop continues with the next column. If no column works, the function returns 0 and the row above tries its next column.

There is no explicit line in the code that undoes the choice. The reason is the representation: `q[r]` is overwritten at the next attempt, and `safe` looks only at rows smaller than *r*. Values left over from a failed attempt are never read. The undoing is done implicitly, by `r` being kept in the frames on the stack.

3.10.3 Code trace: 4 queens

The call `place(0, 4)` proceeds as in Table 3.12. The squares are written in the form *(row, column)* and rows and columns start from 0.

Step	Row	Columns tried	Result	`q`
1	0	0	*(0, 0)* safe	[0, –, –, –]
2	1	0 ×, 1 ×, 2	*(1, 2)* safe	[0, 2, –, –]
3	2	0 ×, 1 ×, 2 ×, 3 ×	no column, `return 0`	[0, 2, –, –]
4	1	3	*(1, 3)* safe	[0, 3, –, –]
5	2	0 ×, 1	*(2, 1)* safe	[0, 3, 1, –]
6	3	0 ×, 1 ×, 2 ×, 3 ×	no column, `return 0`	[0, 3, 1, –]
7	2	2 ×, 3 ×	no column, `return 0`	[0, 3, –, –]
8	1	—	no columns left, `return 0`	[0, –, –, –]
9	0	1	*(0, 1)* safe	[1, –, –, –]
10	1	0 ×, 1 ×, 2 ×, 3	*(1, 3)* safe	[1, 3, –, –]
11	2	0	*(2, 0)* safe	[1, 3, 0, –]
12	3	0 ×, 1 ×, 2	*(3, 2)* safe	[1, 3, 0, 2]
13	4	—	$r = n$: solution, `return 1`	[1, 3, 0, 2]

Table 3.12: Trace of `place(0, 4)`.

The first part of the trace shows that the queen placed in the upper left corner does not lead to a solution. At step 2, column 0 of the second row is in the same column as the first queen and column 1 is on the same diagonal; the first safe column is 2. At step 3 every column of the third row is under threat: columns 0 and 2 fall foul of the column rule, and 1 and 3 of the diagonals of the queen at *(1, 2)*. The function returns 0 and the second row tries its next column, 3.

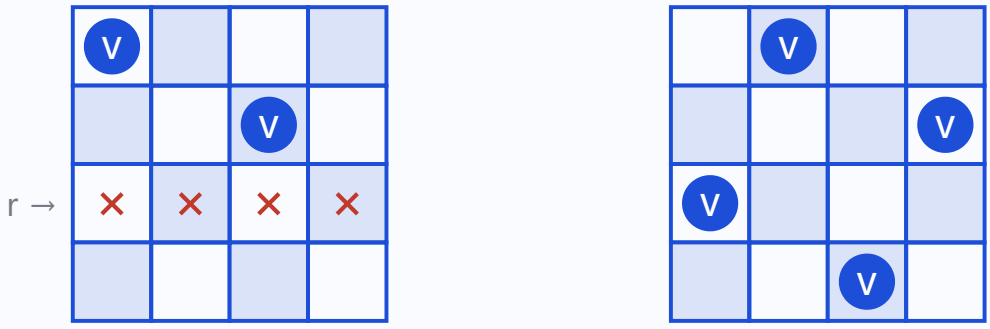


Figure 3.16: 4 queens. (a) Step 3: the queens at *(0, 0)* and *(1, 2)* threaten all squares of the third row (×). (b) Step 13: the solution found, $q = [1, 3, 0, 2]$.

Steps 4–7 show that the second attempt also reaches a dead end. No square is left in the fourth row, and the remaining columns of the third row are under threat too. At step 8 the second row returns 0, having tried its last column. Now the queen of the first row must move. At step 9 the queen is put at *(0, 1)* and this time a safe square is found in every row. At step 13 $r = 4 = n$ and the solution $q = [1, 3, 0, 2]$ is found. The returned 1 passes through four frames and reaches the original caller.

If the assignments `q[r] = c` in the table are counted, eight placements are seen to have been made: *(0, 0)*, *(1, 2)*, *(1, 3)*, *(2, 1)*, *(0, 1)*, *(1, 3)*, *(2, 0)*, *(3, 2)*. Brute force would have tested $4^4 = 256$ candidates. These numbers were verified by adding a counter to `place` and running the program.

Common mistake

Writing the diagonal test in only one direction. The condition `q[i] - c == r - i` catches the diagonal going from upper right to lower left and misses the diagonal in the other direction. Taking the absolute value with `abs` covers both directions at once. Another mistake is to run the loop of `safe` up to $i \le r$; in that case an old value in `q[r]` is read and correct squares may be rejected.

3.10.4 The gain from pruning

An upper bound for the worst-case cost of backtracking is easily found. Because of the column rule, at most $n - r$ columns can be candidates in row r; the columns of the earlier queens are closed. At the first level of the call tree there are at most n nodes, at the second level at most $n(n - 1)$ nodes, and the number of leaves is at most $n!$. The upper bound $O(n!)$ given for the cost comes from here; the `safe` checks made at each node add only a polynomial factor to this class. This is a rough bound: the diagonal rule prunes most of the tree early, and the real cost stays far below $n!$. The stack cost equals the depth, $O(n)$.

n	First solution `q`	Placements	Number of solutions
4	[1, 3, 0, 2]	8	2
5	[0, 2, 4, 1, 3]	5	10
6	[1, 3, 5, 0, 2, 4]	31	4
7	[0, 2, 4, 6, 1, 3, 5]	9	40
8	[0, 4, 7, 5, 2, 6, 1, 3]	113	92

Table 3.13: The first solution, the number of placements, and the total number of solutions for small values of *n*.

The values in Table 3.13 were computed by a program. The placements column counts the assignments `q[r] = c` made up to the first solution. For eight queens, backtracking finds a solution with only 113 placements; compared with the 16.8 million candidates of brute force, the difference is striking. The column for the number of solutions was found by changing the algorithm so that it does not stop at the first solution but counts all solutions. There are 2 solutions for 4 and 92 for 8. The

number of placements does not grow regularly with n: for 5 and 7 the first attempts succeed at once, while for 6 many dead ends are entered. The cost of backtracking depends closely on the structure of the input.

3.11 The knight's tour and Warnsdorff's rule

Backtracking always finds the solution, but slows down as the number of candidates grows. In some problems a good selection rule leads to the solution without any need to go back. The knight's tour is the classic example of this situation.

3.11.1 The problem

Definition 3.11

The **knight's tour**: a chess knight is to visit all the squares of a board, passing through each square exactly once. The knight moves in an L shape: two squares horizontally and one vertically, or two squares vertically and one horizontally.

A knight in the middle of the board can go to eight squares. On the edges and in the corners this number decreases; a knight in a corner reaches only two squares. The moves are kept in a table as the row change Δi and the column change Δj:

Move	0	1	2	3	4	5	6	7
Δi	−2	−1	+1	+2	+2	+1	−1	−2
Δj	+1	+2	+2	+1	−1	−2	−2	−1

Table 3.14: The eight moves of the knight.

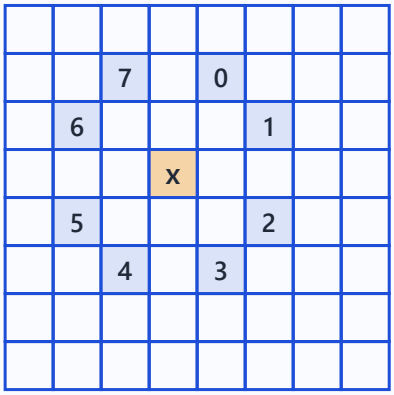


Figure 3.17: The eight squares a knight at x can go to; the numbers are the order in the move table.

The table goes around the moves clockwise: move 0 takes the knight two rows up and one column right, and move 2 takes it one row down and two columns right. Working with the table makes it possible to try all the moves with a single loop instead of writing eight separate `if`s. From the square *(i, j)*, move *k* goes to the square *(i + Δi[k], j + Δj[k])*; the square must be inside the board and not yet visited.

3.11.2 Backtracking and the heuristic

The backtracking solution is similar to n-queens. The knight is put on a square and each move is tried in turn. If no empty square is left for the knight to go to and not all squares have been visited, there is a dead end; the last move is undone and the next move is tried. This method always finds a solution if one exists. But it is brute force in nature. At every square there are up to eight options. In the 64 squares of an 8×8 board a wrong early choice may lead to a dead end noticed much later. The algorithm then has to undo, one by one, the millions of partial tours it has built up to that point.

> **Definition 3.12**
>
> A **heuristic** rests on a rule that makes a choice that looks "clever" at each step. In most cases it finds a good or near-optimal solution, but it does not guarantee this.

Heuristic rules are used in everyday life too. In the game Minesweeper, the player looks at the numbers on the open squares and picks the square that looks safest. This choice is correct most of the time, but sometimes it comes down to luck. A heuristic algorithm works in a similar way: it is fast and usually successful, but inputs on which it fails can be found.

3.11.3 Warnsdorff's rule: a trace on a 5×5 board

The oldest known heuristic rule for the knight's tour is the one proposed by H. C. von Warnsdorff in 1823. The rule says that the next place the knight goes to should be the square whose onward count is nonzero but smallest. The **onward count** of a square is the number of empty squares that can be reached directly from that square.

The logic of the rule is to use up the hard-to-reach squares first. Corner and edge squares can be reached from only a few squares. If the knight moves away without visiting these squares, the paths by which it could come back fill up over time, and the square becomes unreachable. Going to the square with the smallest onward count visits these squares early and does not leave the dead end to the end. The squares in the middle, since they can be reached by many paths, cause no trouble even if they are left for last.

On a 5×5 board a tour starting from the upper left corner, that is, the square *(0, 0)*, can be traced. In case of a tie, the order in the move table (0 to 7) decides. Table 3.15 gives the important steps of the tour. The candidates are the empty squares the knight can go to from that square; the number next to them is the onward count.

Square no.	Position	Candidates (onward count)	Chosen
1	*(0, 0)*	*(1, 2)*: 5, *(2, 1)*: 5	*(1, 2)*, table order in a tie
2	*(1, 2)*	*(0, 4)*: 1, *(2, 4)*: 3, *(3, 3)*: 3, *(3, 1)*: 3, *(2, 0)*: 3	*(0, 4)*
3	*(0, 4)*	*(2, 3)*: 5	*(2, 3)*
4	*(2, 3)*	*(4, 4)*: 1, *(4, 2)*: 3, *(3, 1)*: 2, *(1, 1)*: 3, *(0, 2)*: 3	*(4, 4)*
5	*(4, 4)*	*(3, 2)*: 5	*(3, 2)*
9	*(0, 2)*	*(1, 4)*: 2, *(1, 0)*: 2	*(1, 4)*, table order in a tie
16	*(3, 4)*	*(4, 2)*: 1, *(2, 2)*: 4	*(4, 2)*
24	*(1, 0)*	*(2, 2)*: 0	*(2, 2)*, the last empty square

Table 3.15: Selected steps of the Warnsdorff tour on a 5×5 board.

At the second square the effect of the rule is clearly seen. Among five candidates the upper right corner *(0, 4)* is chosen, because from there only one empty square can be reached. If the corner were not visited now, once the only path leading to it filled up, the corner would remain unreachable forever. At the fourth square the same logic makes the lower right corner, *(4, 4)*, the choice. At the twenty-fourth square only one candidate remains and its onward count is 0. The "nonzero" condition of the rule does not apply here: this square is the last empty square of the board and the tour is completed with it.

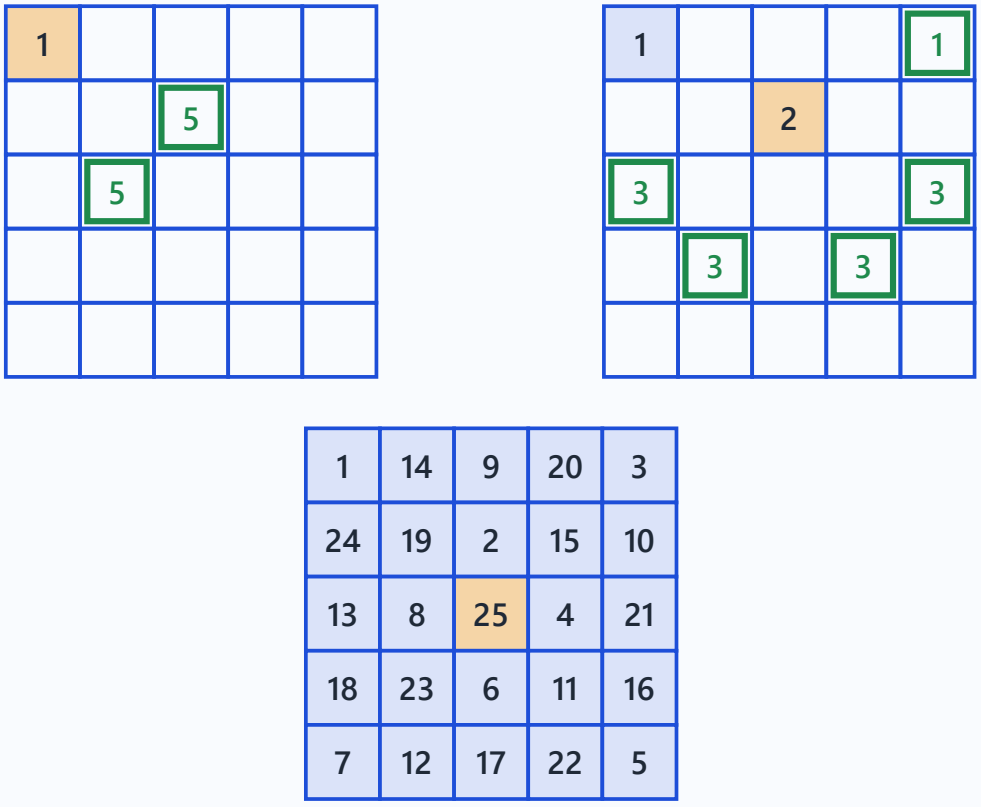


Figure 3.18: The Warnsdorff tour. (a) Square 1: two candidates, both with onward count 5. (b) Square 2: the corner *(0, 4)* has onward count 1 and is chosen. (c) The completed tour: the numbers are the order of visit;

square 25 is the middle of the board.

In the completed tour 25 squares have been visited without ever going back. The order of visit, row by row, is: 1 14 9 20 3 / 24 19 2 15 10 / 13 8 25 4 21 / 18 23 6 11 16 / 7 12 17 22 5. The corners were used up at steps 1, 3, 5, and 7, that is, at the beginning of the tour; the middle square is left to the very end. This tour was produced by applying the rule in a small program, with the candidates and onward counts of every step, and compared with the values in the slides.

The cost of Warnsdorff's rule is very low. On an $n \times n$ board n^2 moves are made. At each move at most 8 candidates are examined, and the onward count of each candidate is computed by looking at at most 8 neighbors. The work per move is constant (at most $8 \times 8 = 64$ square checks); the total work is $O(n^2)$. The memory is $O(n^2)$ for the board itself. Backtracking, on the other hand, tries an exponential number of partial tours in the worst case.

The rule does not always succeed. Depending on how ties are broken and on the starting square, the knight sometimes runs into a dead end. In that case Warnsdorff's rule can be combined with backtracking: the candidates are sorted by onward count and tried in that order, and one goes back at a dead end. Since the rule only determines the order of the attempts, the guarantee of a solution is preserved. In practice going back is very rarely needed.

3.12 Complexity summary

The time and memory costs of the algorithms examined in these notes are given together in Table 3.16. The memory column is the total of the call stack and the extra structures the algorithm uses; the input itself is not counted.

Algorithm	Time	Stack / extra memory	Remark
`fact` **(recursive)**	$\Theta(n)$	$O(n)$	the tail version can be reduced to $O(1)$
Binary search (iterative)	$O(\log n)$	$O(1)$	the array must be sorted
Binary search (recursive)	$O(\log n)$	$O(\log n)$	tail recursion
Hanoi	$\Theta(2^n)$	$O(n)$	$2^n - 1$ moves, optimal
Permutations	$\Theta(n \cdot n!)$	$O(n)$	$n!$ arrangements
Subset generation	$\Theta(n \cdot 2^n)$	$O(n)$	2^n subsets
SAT (brute force)	$O(2^n \cdot \lvert F\rvert)$	$O(\lvert F\rvert)$	$\lvert F\rvert$: size of the formula tree
n-queens (backtracking)	$O(n!)$ upper bound	$O(n)$	pruning tries far fewer in practice
Knight's tour (Warnsdorff)	$O(n^2)$	$O(n^2)$	heuristic; backtracking is exponential

Table 3.16: Complexity of the algorithms in the chapter.

The table falls into two groups. The algorithms in the first three rows are polynomial-time and run comfortably on large inputs. Most of the rest are exponential or factorial time. In Hanoi, permutation, and subset generation this cost is unavoidable, because the output itself has that size. In SAT and n-queens the output is small (an assignment or a placement), but the best known general methods for finding it take exponential time. Warnsdorff's rule is an example of reducing an exponential search to polynomial time with a heuristic choice; in return it gives up the guarantee of a solution.

The stack column has a notable regularity: in all the recursive algorithms the stack cost is the recursion depth. Even when the time is exponential, the depth is often *n*. That is why algorithms like Hanoi or permutation are limited not by stack overflow but by lack of time.

3.13 Common mistakes

1. **Forgetting the base case or writing a step that never reaches it.** The calls never stop and the program crashes with a stack overflow. Every recursive function must have a measure that decreases at every call and a base case that catches the smallest value of that measure.
2. **Not returning the result of the recursive call.** If `binsearch(...)` is written and the returned value is thrown away, the function returns a wrong value even though it found the right answer. A tail call must be written as `return binsearch(...)`.
3. **Using binary search on an unsorted array or a linked list.** In an unsorted array the decision to discard half is wrong and an existing value may not be found. In a linked list reaching the middle takes linear time and the logarithmic gain is lost.
4. **Returning the value of the element found.** The value can be confused with the error marker *−1* and the position information is lost. Binary search must return the index.
5. **Forgetting to undo the choice in backtracking.** If the second `SWAP` in permutations or `pick[i] = 0` in subset generation is not written, sibling branches see a corrupted starting state. Some solutions repeat and some never appear.
6. **Running an exponential algorithm with a large *n* because "the machine is fast".** Hanoi with 64 disks takes hundreds of billions of years, and permutations of 20 elements take decades. The order of growth must be computed before the algorithm is chosen.
7. **Not taking the stack cost of deep recursion into account.** A recursion of depth *n*, if *n* is in the hundreds of thousands, fills a call stack of a few megabytes. Tail recursion is not a guarantee in C; in such cases a loop is preferred.
8. **Overlooking overflow.** `long` is 32 bits on Windows and `fact(13)` overflows. In binary search `(left + right) / 2` can overflow on very large arrays. Signed overflow is undefined behavior and can silently give wrong results.

3.14 Summary

- **Recursion:** a function calling itself with a smaller input. The base case is the smallest instance solved without recursion; the recursive step shrinks the problem and comes closer to the base case at every call.
- **The call stack:** every call opens a stack frame that holds its parameters, local variables, and return address. A recursion of depth d uses $O(d)$ extra memory; the same variable has a separate copy in each frame.
- **Recursion and loops:** every recursion can be turned into a loop with an explicit stack. Recursion is close to the definition; a loop works with $O(1)$ memory and without call cost.
- **Tail recursion:** the recursive call is the last job; it is achieved with an accumulating parameter and the compiler can turn it into a jump. The C standard does not guarantee this optimization.
- **Binary search:** in a sorted array it halves the range in every round and runs at most $\lfloor \log_2 n \rfloor + 1$ rounds: $O(\log n)$. It returns an index, and -1 means "absent". The iterative version uses $O(1)$ and the recursive version $O(\log n)$ extra memory.
- **The towers of Hanoi:** the solution of the recurrence $h(n) = 2h(n - 1) + 1$ is $2^n - 1$, and this number is optimal. The call tree is the history of the stack; the moves are ordered by an inorder traversal.
- **Permutations:** $n!$ arrangements are generated with swap, recursion, and swap back. The number of calls is about $e \cdot n!$ and the time is $\Theta(n \cdot n!)$.
- **Backtracking:** the partial solution is built step by step, a branch is pruned when a rule is broken, and the last choice is undone. Subset generation and n-queens are solved with this method; SAT was solved by brute force, trying all assignments.
- **SAT:** an assignment that makes a logical formula true is sought. Brute force tries 2^n assignments; the formula tree is built once and evaluated with a postorder traversal. The problem is NP-complete.
- **Heuristics:** Warnsdorff's rule takes the knight at each step to the square with the smallest onward count, and the tour is mostly completed in $O(n^2)$ time without ever going back. One never stops looking for a better solution.

3.15 Key terms

recursion recursive base case call stack stack frame stack overflow tail recursion binary search recurrence relation permutation swap brute force backtracking subset satisfiability (SAT) postorder n-queens problem heuristic

3.16 Test Yourself

1. Which calls does `fact_tail(5, 1)` make and what does it return? With tail call optimization off, how many `fact_tail` frames are on the stack at most?

   Answer

   The calls are `fact_tail(5, 1)`, `fact_tail(4, 5)`, `fact_tail(3, 20)`, `fact_tail(2, 60)`, `fact_tail(1, 120)`; the last call is the base case and returns 120. The 120 is passed upward without any operation. With the optimization off, five frames are on the stack at the same time; with it on, a single frame suffices.

2. In the test array (`{4, 13, 18, 19, 24, 27, 33, 36, 39, 40}`), which middle indices does the iterative `binsearch` try when searching for 39, and how many rounds does it run?

   Answer

   The middle indices are 4 (*24 < 39*, `left = 5`), 7 (*36 < 39*, `left = 8`), and 8 (`list[8] = 39`). The function returns 8 in three rounds.

3. How many calls does the recursive `binsearch(list, 5, 0, 9)` make on the same array? With which ranges is it called and what is the result?

   Answer

   Four calls are made: *[0, 9]* (middle 24), *[0, 3]* (middle 13), *[0, 0]* (middle 4), and *[1, 0]*. The last range is empty and *−1* is returned; this value is passed back through three frames. At the deepest moment there are four `binsearch` frames on the stack.

4. How many lines does `hanoi(5, "A", "B", "C")` print, how many times does the function call itself in total (including the first call), and how deep does the stack of `hanoi` frames get at most?

   Answer

   $2^5 - 1 = 31$ lines are printed. Since each call does exactly one `printf`, the number of calls is also 31. The depth equals the number of disks: 5 frames.

5. For `char s[] = "abcd";`, what are lines 7–10 of the output of `perm(s, 0, 4)`? How many calls

are made in total?

Answer

The first six lines begin with a (abcd, abdc, acbd, acdb, adcb, adbc). Then the array becomes `bacd` and lines 7–10 are bacd, badc, bcad, bcda. The number of calls is $1 + 4 + 12 + 24 + 24 = 65$.

6. How many lines does `subsets("abcd", 4, 0, pick)` print, and what is the fifth line? How many times is the function called in total?

Answer

$2^4 = 16$ lines are printed. The order goes {abcd}, {abc}, {abd}, {ab}, {acd}, …; the fifth line is {acd}. The call tree is a perfect binary tree with 5 levels: $2^5 - 1 = 31$ calls.

7. Which assignments (in the order $x_1x_2x_3$) satisfy the formula $(x_1 \vee \neg x_2) \wedge x_3$? How many nodes does the formula tree have?

Answer

$x_3 = 1$ is needed, and either $x_1 = 1$ or $x_2 = 0$: 001, 101, and 111. The tree has six nodes: the ∧ at the root, its left child ∨, the children of ∨, x_1 and ¬, the right child of ¬, x_2, and the right child of the root, x_3.

8. What is the first solution found by the call `place(0, 6)` ? How many solutions are there in all on a 6×6 board?

Answer

The first solution is $q = [1, 3, 5, 0, 2, 4]$; 31 placements are made before it is reached. There are 4 solutions on a 6×6 board; despite there being 10 on 5×5, there are fewer on 6×6.

9. Why can `fact(13)` give a wrong result on Windows? Up to which n does `fact` work correctly on 64-bit Linux?

Answer

On Windows `long` is 32 bits and its largest value is 2,147,483,647. $13! = 6{,}227{,}020{,}800$ exceeds this limit; signed overflow is undefined behavior. On 64-bit Linux `long` is 64 bits; $20! \approx 2.43 \cdot 10^{18}$ fits, $21! \approx 5.1 \cdot 10^{19}$ does not. A correct result is obtained for $n \le 20$.

10. How is a recursive function that finds the largest element of an array written? What is the stack

depth?

Answer

The base case is $n = 1$ and the single element is returned; the recursive step finds the largest of the first $n - 1$ elements and compares it with the last element:

```
int largest(int a[], int n) {
  int m;
  if (n == 1) return a[0];        /* base case */
  m = largest(a, n - 1);          /* largest of the first n-1 */
  return a[n - 1] > m ? a[n - 1] : m;
}
```

Each call decreases n by one; the depth is n and the stack cost is $O(n)$. On the test array the function returns 40. The call is not a tail call, because a comparison is made after the return.

3.17 Further reading

- E. Horowitz, S. Sahni, and S. Anderson-Freed, *Fundamentals of Data Structures in C* (2nd ed.), Section 1.2: the source of the binary search, permutation, and SAT examples in these notes; it gives recursive algorithms and their analyses in the same C language.
- R. Sedgewick, *Algorithms in C*, Chapter 5: describes recursion, divide and conquer, and call trees with detailed drawings.
- T. H. Cormen et al., *Introduction to Algorithms* (3rd ed.), Chapters 4 and 34: methods for solving recurrence relations and the theory of NP-completeness, with the place of SAT in that theory.
- N. Wirth, *Algorithms + Data Structures = Programs* (1976): a classic textbook that builds the general pattern of backtracking in Pascal and illustrates it with problems such as eight queens.

3.18 References

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 3rd ed., MIT Press, 2009 (Chapter 4: recurrences; Chapter 34: NP-completeness, SAT).
2. E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2nd ed., Silicon Press, 2008 (Section 1.2: recursive algorithms).
3. R. Sedgewick, *Algorithms in C*, Addison-Wesley (Chapter 5: recursion).
4. N. Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall, 1976.

PART II

Linear Structures

Chapters 4–9

PART II · CHAPTER 4

Linked Lists

An array keeps its elements side by side in memory. That layout makes indexing very fast, but inserting an element in the middle or deleting one from the middle forces every element behind it to shift. A linked list solves the problem with a different layout. Each element lives in its own node, and each node carries the address of the next one. The order now comes from these addresses rather than from position in memory. The chapter starts with the pointer and dynamic memory machinery that C needs to build such a structure. It then covers traversal, insertion, deletion, search and releasing a singly linked list, each with code traces. The classic routines follow: reversal, concatenation, finding the middle node and Floyd's cycle detection. The last sections treat the circular list, the header node, the doubly linked list and the available-node list that reuses deleted nodes.

LEARNING OUTCOMES

Build a node
Define a self-referential `struct`, allocate a node with `malloc` and check the result, and return a finished node with `free`.

Link discipline
Insert at the front, at the end, in the middle and in sorted position without losing a single link, in the right order; pass the list by address to routines that change the head pointer.

Safe deletion
Delete a node, and release a whole list, without creating a memory leak or a dangling pointer.

Classic routines
Write reversal, concatenation, middle-node search and Floyd's cycle detection using *O(1)* extra memory, and justify their costs.

List variants and memory management
Choose among circular, header-node and doubly linked lists according to the purpose; reuse nodes with an available-node list.

4.1 When arrays are not enough

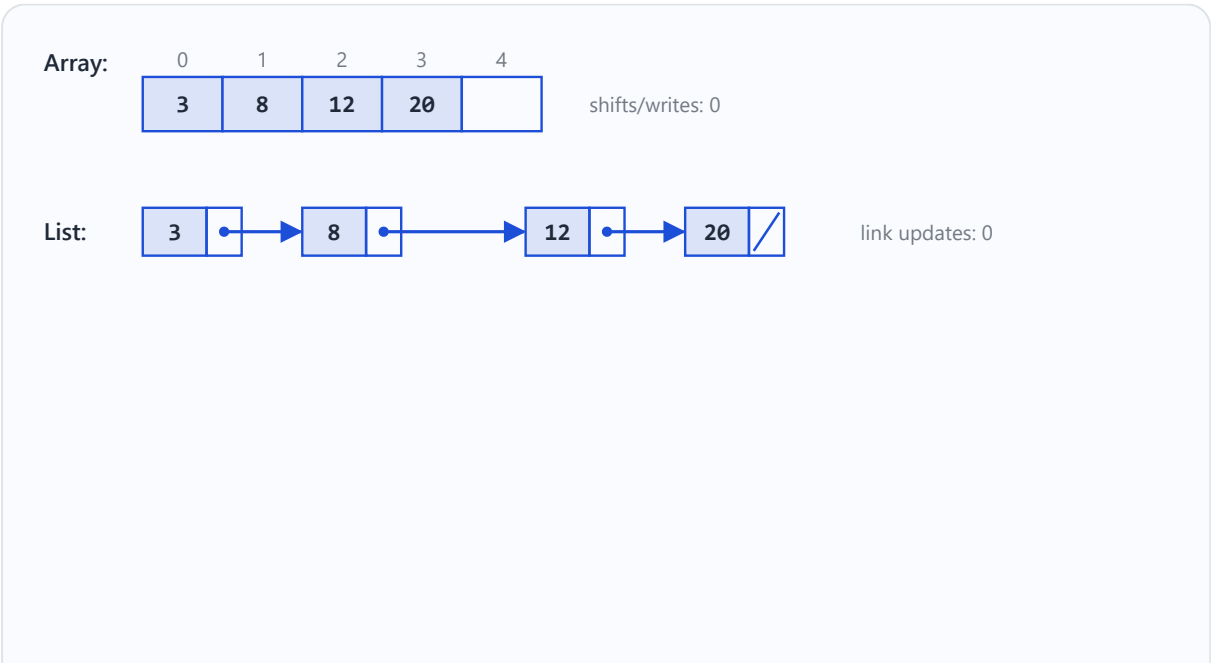
Choosing a data structure starts with the question of which operations will be frequent. An array is excellent at some operations and rather poor at others. A linked list is strong exactly where the array is weak, and weak where the array is strong. This section compares the two structures on the same insertion task and introduces the basic idea of the linked list: the concepts of node and link.

4.1.1 The cost of insertion and deletion in an array

In C, an array is a single block in which elements of the same type are laid out one after another in memory. The declaration `int a[5]` reserves a block of twenty bytes; `a[0]` occupies the first four bytes of the block, `a[1]` the next four. This layout has a great advantage. The address of `a[i]` is found with a single computation: *i · sizeof(int)* is added to the starting address. Reaching any element of an array therefore takes constant time, however long the array is, that is, *O(1)*. This is called **random access**.

The same layout becomes a burden for insertion and deletion. No gaps can be left between elements; order is determined by position. To insert a new value into the middle of a sorted array, room must first be made for it. The only way to make room is to shift every element after the insertion point one cell to the right. Deletion works the other way around: the gap left by the deleted element is closed by shifting the elements behind it one cell to the left.

Figure 4.1 shows the difference with a concrete example. The value 10 is to be inserted into the sorted sequence 3, 8, 12, 20, right after 8. In the array, 20 moves one cell to the right first, then 12 does. Only then can 10 be written into the vacated cell, number 2. In total there are two moves and one write. In the list, no element moves at all. The new node is placed in any free spot in memory and two links are changed: the new node points to 12, and 8 points to the new node.



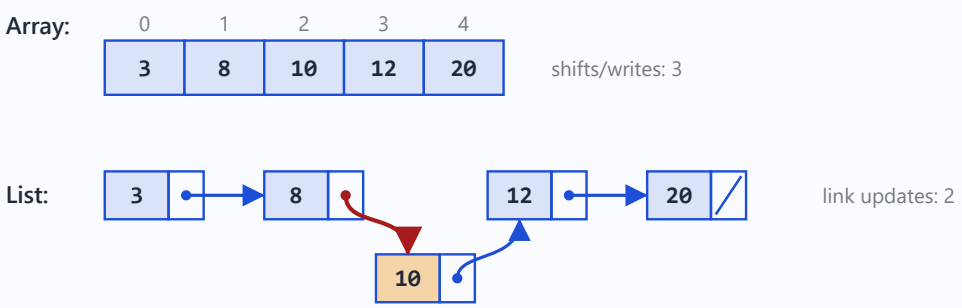


Figure 4.1: Inserting 10 after 8 in the sequence 3, 8, 12, 20, stored as an array and as a list. The upper drawing is the initial state. In the lower drawing the array has shifted 20 and 12 one cell to the right and written 10 into the vacated cell (three shifts/writes). In the list the new node points to 12 and 8 points to the new node; only two links have been updated.

This example has four elements, and the difference looks small. But the number of elements an array must shift equals the number of elements after the insertion point. If an element is inserted at the front of an array of n elements, all n elements are shifted. So insertion and deletion in an array cost $O(n)$ in the worst case. In a list, if the node at which the insertion is to be made is known, only two links are written. This work does not change however long the list is; the cost is $O(1)$.

The second weakness of the array is its size. The size of a static array is fixed at compile time. A dynamic array can be grown with `realloc`, but growing may require copying all elements into a new block. A linked list, on the other hand, asks for a separate small block for each element. The list grows by one node and shrinks by one node; no capacity has to be guessed in advance.

The linked list has its price as well. Because nodes are scattered in memory, the address of the i-th node cannot be computed. The only way to reach it is to start at the first node and follow the links one by one. Access to the i-th element of a list therefore takes $O(i)$. In addition, each node carries an address next to its data, which means extra memory. In short, the two structures are opposites. If access is frequent and changes are rare, the array is the better fit; if insertions and deletions are frequent, the linked list is.

4.1.2 Nodes and links

Definition 4.1

A **linked list** is a linear data structure that keeps its elements in separate blocks of memory called **nodes**. Each node consists of two fields: a **data field** that holds the value of the element and a **link** field that holds the address of the next node. The link field of the last node contains `NULL`. The pointer that holds the address of the first node is called the **head** of the list. The **empty list**, which has no nodes at all, is the list whose head is `NULL`.

In this definition each node knows only its successor. Such a list is called a **singly linked list**. The fact that a node does not know the node before it shapes the form of many operations. For example, to delete a node one must change the link of the node before it, but in a singly linked list there is no way to go back to the predecessor.

A linked list can be compared to a treasure hunt. Each clue slip carries a value and says where the next slip is hidden. It does not matter where in the house the slips lie; the order is established by the directions written on them. Someone who knows where the first slip is can find all the slips in order. Someone who forgets where the first slip is cannot reach any of them, even though they are all still in the house. Losing the head pointer of a linked list is exactly this: the nodes are in memory, but no variable points to them any more.

4.1.3 The view in memory

The most important property of a linked list is that its nodes do not have to sit next to each other in memory. Figure 4.2 gives both the actual layout in memory and the logical view of a list of three nodes. On the left, memory is drawn as a strip together with its addresses. The node with value 12 is at address `0x1A40`, the node with value 7 at `0x2C10`, and the node with value 25 at `0x0F88`. Other data lie in between.

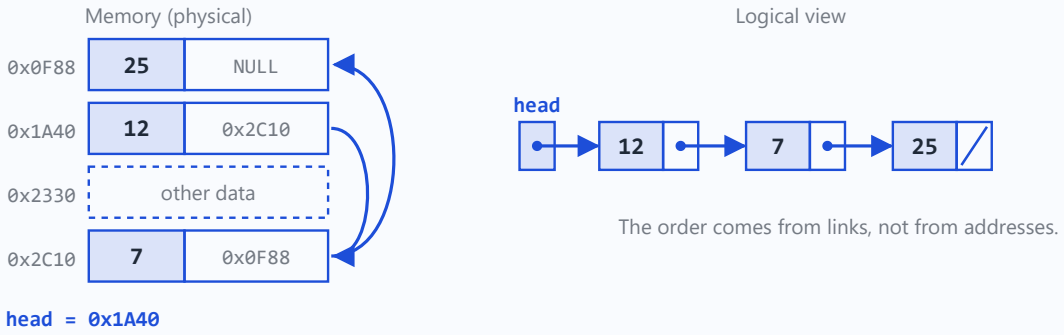


Figure 4.2: Nodes are scattered in memory. The variable `head` holds the address of the first node (`0x1A40`). The link field of 12 holds the address of 7 (`0x2C10`), the link field of 7 holds the address of 25 (`0x0F88`), and the link field of 25 holds `NULL`. The logical view on the right draws the same list with boxes and arrows.

Looking at the addresses, the order appears as 0x0F88, 0x1A40, 0x2C10. If the list were read in that order it would be 25, 12, 7. Yet the order of the list is 12, 7, 25. The source of the order is not the magnitude of the addresses but the links: `head` points to 12, 12 points to 7, and 7 points to 25. The logical view on the right side of the figure draws this information with boxes and arrows. All drawings used throughout these notes follow this box-and-arrow notation. The left half of a box shows the data field and the right half shows the link field. The arrow leaving a link field goes to the node whose address is in that field. A slash in a link field means `NULL`.

The arrows in the drawings are a convenience. There are no arrows in memory; there are only numbers. A link field stores the starting address of the other node as an integer. Going to the node an arrow "points to" means using that number as an address and reading the memory there. This distinction matters for understanding pointer errors. For example, when a node is returned with `free`, the number in any other link that points to it is not erased automatically. In the drawing the arrow is still there, but it now points to an invalid place.

The notion of an address has also changed over time. In old x86 processors an address consisted of two parts: a segment and a position within that segment. This arrangement is called **segmented memory**, and in the C compilers of that era it gave rise to distinctions such as "near" and "far" pointers. In today's operating systems every program has a single, flat address space. A pointer in this space is nothing but a number. On a 64-bit system a pointer takes 8 bytes.

4.2 Pointers and dynamic memory in C

The linked list is built on two tools of C: the pointer, which holds the address of another object, and `malloc` and `free`, which let a program request memory at run time and give it back. Unless these two tools are well understood, linked list code never goes beyond rote memorization. If this topic is not mastered in the second year, it tends to stay incomplete in the years that follow; this is why the course returns to C and pointers before moving on to linked lists. It is quite normal for a student who believes they know pointers to get stuck on list code anyway. The section below therefore rebuilds the basic concepts briefly but completely.

4.2.1 Addresses and pointers

Definition 4.2

A **pointer** is a variable that holds the memory address of an object. The declaration `T *p` defines a pointer `p` that can hold the address of an object of type `T`. The expression `&x` gives the address of `x`. The expression `*p` is the object that `p` points to; it can be read and overwritten. The value of a pointer that points to no object is `NULL`.

The short code fragment from the slides brings together these operators and the allocation and release of a node:

```c
int x = 5;
int *p = &x;       /* p holds the address of x */
*p = 7;            /* write through p: x == 7 */

node_ptr q = NULL; /* no node yet */
q = malloc(sizeof *q);
if (q != NULL) {
    q->data = 3;   /* (*q).data = 3 */
    q->link = NULL;
}
free(q);           /* returned; q now dangles */
q = NULL;
```

The first three lines involve only an integer and a pointer. `int x = 5;` reserves four bytes of memory and writes 5 there. The line `int *p = &x;` defines a pointer named `p` and puts the address of `x` into it. Now `p` and `x` are two separate variables: `x` holds a number, `p` holds an address. The line `*p = 7;` writes 7 to the place `p` points to, that is, to `x`. The value of `x` has changed although `x` was never mentioned by name. This is also how a function can modify its caller's variable: the function is given the address of the variable, not the variable itself.

The `*` sign does two different jobs in two different places, and this is a common source of confusion for beginners. In a declaration (`int *p`) it means " `p` is a pointer". Inside an expression (`*p = 7`) it means "the object `p` points to" and is called the **dereference operator**. The `&` operator is the **address-of operator**. The two are inverses of each other: `*&x` is `x` itself.

Line	`x`	`p`	`q`	Explanation
`int x = 5;`	5	—	—	4 bytes were reserved for `x`.
`int *p = &x;`	5	&x	—	`p` holds the address of `x`.
`*p = 7;`	7	&x	—	`x` was written through `p`.
`node_ptr q = NULL;`	7	&x	NULL	No node yet.
`q = malloc(sizeof *q);`	7	&x	0x2C10	A 16-byte block was allocated on the heap; its fields hold garbage values.
`q->data = 3; q->link = NULL;`	7	&x	0x2C10	The node: `data = 3`, `link = NULL`.
`free(q);`	7	&x	0x2C10	The block was returned; `q` still holds the old address (dangling).
`q = NULL;`	7	&x	NULL	The dangling value was erased.

Table 4.1: Trace of the pointer fragment. The addresses are sample values; real addresses change with every run.

4.2.2 Pointers to structures and the `->` operator

The second half of the fragment works on a node. `node_ptr` is the "pointer to node" type, which will be defined in the next section. Since `q` holds the address of a node, reaching the `data` field of the node means going first to the node itself with `*q` and then to the field with the dot operator: `(*q).data`. The parentheses here are mandatory. In C the dot operator has higher precedence than `*`. Without parentheses, the expression `*q.data` is read as `*(q.data)`; since `q` is a pointer and not a structure, this expression does not compile.

Because this notation is both long and error-prone, C offers a shorthand: `q->data`. The arrow operator `->` goes to the field on its right in the structure that the pointer on its left points to. `q->data` and `(*q).data` are exactly the same. In linked list code the arrow operator is chained constantly. `p->link->data` is the value of the node after the one `p` points to; `p->link->link` is the address of the node two steps ahead. When reading such a chain, one must be sure that the pointer in front of each arrow is not `NULL`. Accessing a field through a `NULL` pointer is undefined behavior and on most systems ends in a program crash.

4.2.3 Allocating a node with `malloc`

Local variables defined inside a function disappear when the function returns. The nodes of a linked list, however, must outlive the function that creates them. This is why nodes are allocated from **heap memory**. A block in heap memory is requested with `malloc` and lives until the program gives it back with `free`. `malloc` allocates a block of the requested number of bytes and returns the starting address of the block as type `void *`. In C, `void *` converts to any pointer type automatically, so it is enough to assign the return value without a **cast**. If no memory is left, `malloc` returns `NULL`.

The slides gather node allocation into a single helper function:

```
node_ptr new_node(int x) {
    node_ptr p = malloc(sizeof *p);
    if (p == NULL) {
        fprintf(stderr, "Out of memory...\n");
        exit(1);
    }
    p->data = x;
    p->link = NULL;
    return p;
}
```

- `malloc(sizeof *p)` asks for as much space as the type `p` points to, that is, `struct node`. `sizeof *p` is computed at compile time; the current value of `p` is not looked at. It is therefore safe even when `p` has not yet been given a value. Not repeating the type name prevents the size from going wrong if the type of `p` changes later.

- The return value is checked against `NULL` immediately. Without the check, when memory runs out the next line would try to write to a `NULL` address. The reaction chosen here is to write a message to the standard error stream and end the program with `exit(1)`. The message in the slides announces the situation in a humorous tone.
- The fields of the new node come from `malloc` with garbage values. The assignments `p->data = x` and `p->link = NULL` bring the node to a consistent state: the data is written and the node is not yet linked anywhere.
- The function returns the address of the node. Because the node lives in heap memory, it does not disappear when the function returns; only the local variable `p` does.

How much space does a single node take? On a 64-bit Windows or Linux system an `int` is 4 bytes and a pointer is 8 bytes. Because a pointer must sit at an address that is a multiple of 8, the compiler inserts 4 bytes of padding after the `data` field. As a result `sizeof(struct node)` is 16 bytes (this value was measured in the verification program of these notes). Sixteen bytes are spent to store four bytes of data. On top of that, `malloc` keeps a few more bytes of its own bookkeeping beside each block. This is why the memory cost of a linked list is noticeable for small elements.

At this point a very common misunderstanding must be corrected. It is not necessary to call `malloc` for every pointer variable. `malloc` is called only when a *new node* is needed. Pointers such as `p` that are used to walk over the list point to nodes that already exist; allocating space for them is both unnecessary and a cause of memory leaks.

Common mistake

Allocating space for a traversal pointer: `node_ptr p = malloc(sizeof *p); p = list;`. The first line allocates a node, and the second line overwrites `p` with the head of the list. The address of the allocated block is now in no variable; the block occupies space for nothing until the program ends. The correct code is simply `node_ptr p = list;`.

4.2.4 `free` and memory errors

`free(p)` returns the block that `p` points to back to memory. The rule is simple: for every address returned by `malloc`, `free` is called exactly once. Breaking this rule in the two possible directions gives two different kinds of error.

Definition 4.3

A **memory leak** is memory that no pointer can reach any more but that has not been returned with `free`. A **dangling pointer** is a pointer that keeps holding the address of an object that has been freed or whose lifetime has ended.

A leak comes from never calling `free`. In a short-running program it may go unnoticed, because the operating system takes all memory back when the program ends. But in a server that runs continuously, or in a program that builds and abandons lists inside a loop, the leaked memory accumulates and eventually memory runs out. A dangling pointer comes from the opposite cause, from returning memory too early. The call `free(q)` does not change the value of `q`; `q` keeps holding the old address. Reading from or writing to this address is undefined behavior. `malloc` may have handed the returned block out again for another purpose; in that case the value written corrupts completely unrelated data of the program. Freeing the same address a second time is also undefined behavior and can corrupt the internal bookkeeping of `malloc`.

This is why the last two lines of the fragment from the slides are written together: first `free(q)`, immediately followed by `q = NULL`. The second line is not mandatory, but it is a good habit. Using a `NULL` pointer by mistake crashes the program at once on most systems, and the bug shows immediately. Using a dangling pointer, on the other hand, often “works” silently, and the bug appears much later in an unrelated place. In addition, according to the C standard `free(NULL)` does nothing, so releasing a pointer that has been set to `NULL` a second time by mistake is harmless.

Common mistake

Reading a field of a node after freeing it: `free(p); p = p->link;`. The second line reads from a block that has been returned. The program often gives the expected result, because the block has not yet been given to another use; but this is luck, not correctness. The right order is to save the address that is needed first and free afterwards: `q = p->link; free(p); p = q;`. This rule shows up in all of the deletion and release code of this chapter.

4.3 Singly linked list: nodes and traversal

This section builds the definition of a linked list in C and describes the traversal pattern that visits every node of the list in order. Traversal is the skeleton of almost all list operations, including insertion and deletion.

4.3.1 The self-referential structure

A node carries two fields: an integer and the address of the next node. The type of the second field is “pointer to node”. In other words, the node structure contains a field that points to an object of its own type. Such a structure is called a **self-referential structure**. The slides define this structure in the form used in Horowitz's book, first giving a name to the pointer type:

```c
typedef struct node *node_ptr;  /* the pointer type first */

struct node {
    int data;         /* data field */
    node_ptr link;    /* address of the next node */
};

node_ptr list = NULL;  /* the empty list */
```

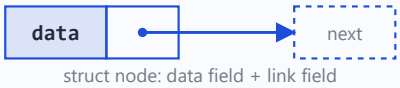


Figure 4.3: A node: the `data` field and the `link` field. The arrow leaving `link` points to the next node (the dashed box).

The first line gives the pointer type the name `node_ptr` (node pointer) before `struct node` has been defined. This line compiles, because C allows a pointer to a structure whose size is not yet known to be defined. Such a structure is called an **incomplete type**. A variable of an incomplete type cannot be defined, because the compiler does not know how much space to give it. But a pointer to an incomplete type can be defined, because the size of every pointer is known regardless of the type it points to. The `struct node` definition that follows completes the structure. Its `link` field is of the `node_ptr` type just defined.

The same structure can also be written without `typedef`: `struct node { int data; struct node *link; };`. In this form the structure can already refer to its own name inside its definition. The choice of name is a matter of habit. In books and old exam questions one sees names such as `list_pointer`, `list_ptr` or `node_ptr` for the same type. These notes use `node_ptr`, consistent with the slides; `list_ptr` appearing in an exam question is the same type.

The last line, `node_ptr list = NULL;`, creates an empty list. There is no separate type for the linked list itself. A list is represented by a single pointer that holds the address of the first node. If this pointer is `NULL` the list is empty. Every operation must therefore consider the empty list separately.

Common mistake

Writing `typedef struct list { … };` and not putting a type name at the end. `typedef` gives a new name to a type; if no name is written the declaration is wasted, and most compilers warn “typedef requires a name”. Either the word `typedef` must be dropped or the type name must be written after the closing brace: `typedef struct list { … } list_t;`.

4.3.2 Traversal

Visiting all nodes of a list in order from beginning to end is called **traversal**. The only tool of traversal is a pointer. The pointer starts at the first node, at each step reads the link field of the node it is on to move to the next one, and stops when it reaches `NULL`. The function that prints all values of the list is the simplest example of this pattern:

```
void print_list(node_ptr p) {
    for (; p != NULL; p = p->link)
        printf("%d ", p->data);
    printf("\n");
}
```

The first part of the `for` loop is empty, because `p` already arrives as a parameter holding the address of the first node. The condition `p != NULL` keeps the loop going as long as `p` points to a valid node. The body prints the value of that node. The increment part `p = p->link` moves `p` to the next node. In a linked list this assignment is the counterpart of `i++` in an array. Because the link of the last node is `NULL`, after the last node has been printed `p` becomes `NULL` and the loop ends.

There is an important detail in the call `print_list(list)`. In C all arguments are passed **by value**. The function receives not the variable `list` itself but a copy of the address inside it. The function advances this copy (`p`) in the loop and finally makes it `NULL`. The `list` in `main` never changes and still points to the first node. Doing the traversal directly on the parameter is therefore safe. If the same loop were written inside `main` on the variable `list`, the head of the list would be lost.

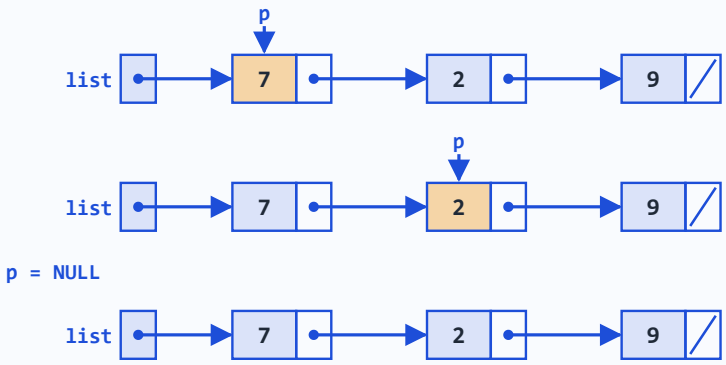


Figure 4.4: Traversing the list 7 → 2 → 9 with `print_list`. At the top, `p` is on the first node and 7 is printed. In the middle, after `p = p->link`, it is on the second node and 2 is printed. At the bottom the link of the third node has been followed and `p` has become `NULL`; the loop ends.

Figure 4.4 shows three moments of the traversal of the list 7 → 2 → 9. The loop runs three times and the output is `7 2 9`. Each iteration does a constant amount of work: one comparison, one print and one assignment. On a list of n nodes the loop runs exactly n times, and the condition is tested $n + 1$

times. The running time of traversal is therefore $\Theta(n)$. The only extra memory is the pointer `p`, that is, $O(1)$.

Finding the length of a list uses the same pattern; instead of printing, a counter is incremented: `for (; p != NULL; p = p->link) n++;`. In an array the length is either stored in a variable or known at compile time. In a linked list, unless the length is kept separately, the whole list must be traversed every time it is asked for, which takes $\Theta(n)$. In applications where the length is needed often, keeping a counter alongside the list is therefore a common solution.

Common mistake

Writing the loop condition as `p->link != NULL`. This condition ends the loop on arriving at the last node, so the value of the last node is never printed. Worse, if the list is empty, `p` is `NULL` and the expression `p->link` crashes the program. The condition `p->link != NULL` is correct only when the goal is to *find* the last node; even then the empty list must be checked separately.

4.4 Insertion

There are four ways to insert a node into a linked list: at the front, at the end, after a known node, and at the right place in a sorted list. In all four the job amounts to allocating a new node and writing two links in the right order. The difference lies in how the place of insertion is found and in whether the head of the list changes.

A single rule holds for every insertion: **save first, then link**. Before a link field is overwritten, the address it holds must have been kept somewhere else. Otherwise all the nodes behind that address are lost. This rule is the reason why, in every piece of code below, the link of the new node is written *before* the link in the list.

4.4.1 Inserting at the front and passing by address

Insertion at the front is the simplest insertion: the new node points to the old first node, and the new node becomes the head of the list. Simple as it is, it raises an important question specific to C. The head of the list is a variable in `main`, and the function must modify that variable. The code in the slides therefore takes the list by address:

```
bool push_front(node_ptr *head, int x) {
    node_ptr fresh = malloc(sizeof *fresh);
    if (fresh == NULL) return false;
    fresh->data = x;
    fresh->link = *head;
    *head = fresh;
    return true;
}

node_ptr list = NULL;
push_front(&list, 2);   /* 2 */
push_front(&list, 7);   /* 7 -> 2 */
```

The first parameter of the function is `node_ptr *head`, that is, a “pointer to pointer to node”. The call writes `&list`, handing the function the address of the variable `list`. Inside the function, `*head` is the variable `list` of `main` itself. Reading `*head` reads `list`, and writing to `*head` writes to `list`. This technique is called **passing by address**, and `head`, which holds the address of a pointer, is called a **pointer to pointer**.

What would happen if the parameter were taken by value, as `node_ptr head`? The function would receive a copy of `list`. The assignment `head = fresh` would change only this local copy. When the function returned, the copy would vanish, while `list` in `main` would keep pointing to the old first node. The new node would have been allocated and correctly linked, but no variable could reach it: a node that is neither inserted nor freed, a leak. The compiler does not catch this error, because the code is entirely valid syntactically.

A good rule helps here. A variable that appears as `*head` in a routine's parameter list is also used as `*head` inside the routine. If `head` shows up bare, without the star, either something deliberate is going on or there is a bug. This simple check catches most pass-by-address errors while reading.

The body of the function consists of five steps. A new node is requested with `malloc`. If memory could not be allocated, the function returns `false`; the list has not changed and is not damaged. Then the value is written. The line `fresh->link = *head` saves the old head: the new node points to the node that has been first until now. Finally `*head = fresh` changes the head. The `bool` return type and the values `true` / `false` come from the `<stdbool.h>` header.

The order of the last two lines is vital. If `*head = fresh` were written first, the head of the list would be the new node. The following `fresh->link = *head` would then write the address of the new node into its own link. The new node would point to itself and the whole old list would be lost. The only variable that pointed to it was `*head`, and it has been overwritten.

The cost of insertion at the front does not depend on the length of the list. One `malloc`, one comparison and three assignments are performed. The time is *O(1)*, and the extra memory is the new node itself. The example in the slides starts from the empty list: `push_front(&list, 2)` makes the

list 2, and `push_front(&list, 7)` makes it 7 → 2. Insertion at the front reverses the order; the most recently inserted element stands first. In the empty list `*head` is `NULL` and `fresh->link` becomes `NULL`, so no separate branch is needed for the empty list.

4.4.2 Code trace: `push_front(&list, 5)`

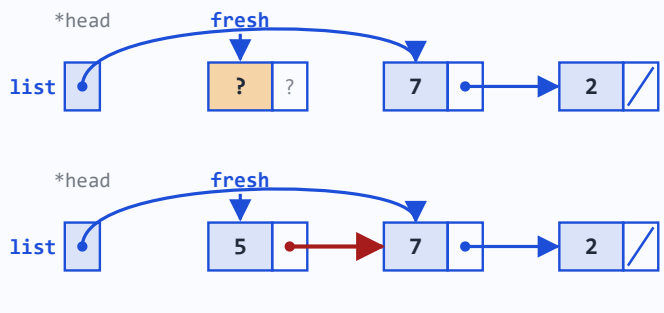
The slides run the same function step by step, with the comments spelled out:

```
bool push_front(node_ptr *head, int x) {
    node_ptr fresh = malloc(sizeof *fresh);
    if (fresh == NULL) return false;  /* out of memory */
    fresh->data = x;
    fresh->link = *head;   /* save the old head first */
    *head = fresh;         /* then change the head */
    return true;
}
```

Initially the list is 7 → 2. Table 4.2 gives the state of memory after each line; Figure 4.5 draws the three most important moments.

Line	`fresh`	`fresh->data`	`fresh->link`	`*head` (= `list`)	List
Call	?	—	—	&node(7)	7 → 2
`fresh = malloc(…)`	&new node	? (garbage)	? (garbage)	&node(7)	7 → 2
`if (fresh == NULL)`	False: memory was allocated, continue.				
`fresh->data = x`	&new node	5	? (garbage)	&node(7)	7 → 2
`fresh->link = *head`	&new node	5	&node(7)	&node(7)	7 → 2 (the new node points to 7)
`*head = fresh`	&new node	5	&node(7)	&node(5)	5 → 7 → 2
`return true`	The function returns; `list` in `main` now points to 5.				

Table 4.2: Trace of `push_front(&list, 5)`; initially the list is 7 → 2.



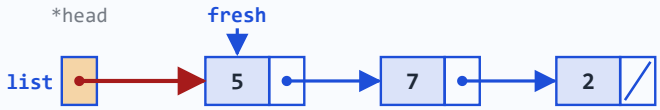


Figure 4.5: Three moments of insertion at the front. At the top, `malloc` has allocated the new node; its fields hold garbage values (?). In the middle, the value has been written and `fresh->link = *head` makes the new node point to 7; `list` is still at 7. At the bottom, `*head = fresh` moves `list` in `main` to 5: 5 → 7 → 2.

The fifth row of the table contains an interesting intermediate state. The new node points to 7, but the list still starts at 7. At this moment the contents of the list have not changed; the new node has merely been prepared. The structure of the list changes with a single assignment, in the line `*head = fresh`. Such an arrangement is also safe in the failure case. Had `malloc` failed, the function would have returned at the first check and the list would have been left untouched.

4.4.3 Inserting at the end

When inserting at the end, the new node will be the last node, so its link is `NULL`. The real work is to find the current last node and turn its link toward the new node. Since the list is held only by its head, the last node can be found only by walking from the beginning:

```
bool push_back(node_ptr *head, int x) {
    node_ptr p, fresh = malloc(sizeof *fresh);
    if (fresh == NULL) return false;
    fresh->data = x;
    fresh->link = NULL;             /* the new last node */
    if (*head == NULL) { *head = fresh; return true; }
    for (p = *head; p->link != NULL; p = p->link)
        ;                           /* find the last node */
    p->link = fresh;
    return true;
}
```

- The first two lines prepare the new node: allocate it, check it, write its value and set its link to `NULL`.
- If the list is empty (`*head == NULL`) there is no last node. The new node becomes both the first and the last node, and the head changes. Because of this case, this function too takes the list by address.
- If the list is not empty, the `for` loop starts `p` at the head and advances it as long as `p->link != NULL`. The body of the loop is empty; the lone semicolon makes this explicit. When the loop ends, `p` points to the node whose link is `NULL`, that is, the last node.
- `p->link = fresh` attaches the new node to the end of the chain.

The loop condition here differs from the one in traversal: it is `p->link != NULL`, not `p != NULL`. The aim is not to step off the list but to stop at the last node. This condition assumes that `p` is not `NULL`; that is why the check for the empty list is done before the loop.

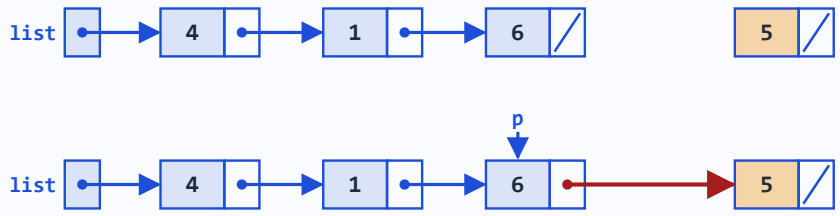


Figure 4.6: Inserting 5 at the end of the list 4 → 1 → 6. At the top, the new node is ready and its link is `NULL`. At the bottom, `p` has found the last node (6) and `p->link = fresh` has been written: 4 → 1 → 6 → 5.

The cost of insertion at the end comes from the walk that finds the last node. On a list of n nodes the loop takes $n - 1$ steps; the time is $\Theta(n)$. Insertion into the empty list is $O(1)$. Building a list of n nodes from empty by inserting at the end therefore takes $0 + 1 + 2 + \ldots + (n - 1) = n(n - 1)/2$ steps, that is, $\Theta(n^2)$. This cost is unacceptable in applications where insertion at the end is frequent.

The solution is simple: keep a second pointer (`tail`) beside the head that holds the address of the last node. Since the last node is known at every moment, insertion at the end becomes $O(1)$: `tail->link = fresh; tail = fresh;`. The price is keeping this pointer up to date in every operation. For example, if the last node is deleted, `tail` must be updated as well. The linked queue is implemented in exactly this way: elements are added at the end and removed from the front, and both operations take $O(1)$. A circular list obtains the same gain with a single pointer.

4.4.4 Inserting in the middle

If the node immediately before the place of insertion is known, insertion needs no walk. The function in the slides inserts a new node with value `x` after the node `prev`:

```
/* inserts a new node with value x after prev */
node_ptr insert_after(node_ptr prev, int x) {
    node_ptr p = new_node(x);  /* malloc + check */
    if (prev != NULL) {      /* middle or end */
        p->link = prev->link;
        prev->link = p;
    }                          /* else stand-alone */
    return p;
}
```

The new node is allocated with `new_node`; the memory check lives inside that function. If `prev` is not `NULL`, two links are written. The line `p->link = prev->link` copies into the new node the address of the node that `prev` has been pointing to. This is the "save first" step. `prev->link = p` then hooks the new node into the chain. If `prev` is the last node, `prev->link` is `NULL` and the new node is added at the end; the same code covers both cases. If `prev` is `NULL`, the new node is not attached to any list and is returned on its own. In that case the caller is expected to use the returned node as the head of a new list.

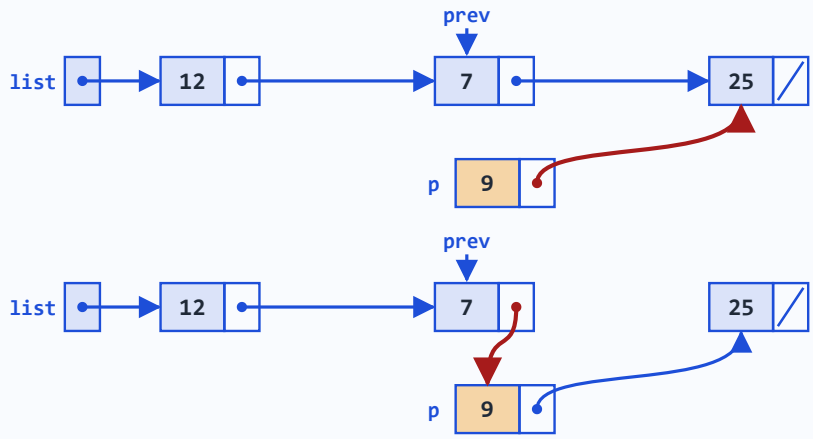


Figure 4.7: Inserting 9 after 7 in the list 12 → 7 → 25. At the top, `p->link = prev->link` has copied the address of 25 into the new node; two links now point to 25 at once. At the bottom, `prev->link = p` makes 7 point to the new node: 12 → 7 → 9 → 25.

The upper drawing of Figure 4.7 shows an important intermediate moment: 7 and 9 point to 25 at the same time. This is not a mistake; it is perfectly allowed for several links to point to one node. At this moment the list is still 12 → 7 → 25, because someone traversing from the head never visits 9. After the second assignment, 9 enters the chain. If the order of the assignments were reversed, that is, if `prev->link = p` were written first, the address of 25 would be lost. The following `p->link = prev->link` would then write the node's own address into its link, and 9 would turn into a node pointing at itself.

The function writes only two links and contains no loop; its time is *O(1)*. How `prev` is found is a separate question. If it has to be found by value, a search is needed and the total cost becomes the *O(n)* of the search. This function is a generalization of the routine used in polynomial multiplication, which appends a new term after the previous term: there too the new term is added after the previous one, and since the previous term is known at every step, the insertion takes constant time.

4.4.5 Sorted insertion

When a new value is inserted into a list whose values are kept in ascending order, the new node must be put in the right place so that the order is not broken. The right place is immediately after the last node that is smaller than the new value. In a singly linked list, inserting after a node requires that

node itself. For this reason two pointers walk together during the search: `p` points to the node being examined and `prev` to the node one behind `p`.

```c
void sorted_insert(node_ptr *head, node_ptr fresh) {
    node_ptr prev = NULL, p = *head;
    while (p != NULL && p->data < fresh->data) {
        prev = p;
        p = p->link;
    }
    fresh->link = p;
    if (prev == NULL) *head = fresh;  /* at the front */
    else prev->link = fresh;          /* middle or end */
}
```

The function receives the new node ready-made; allocating the node and writing its value is the caller's job. `prev` starts at `NULL` and `p` at the first node. The `while` loop advances the two pointers together one step as long as `p` has not left the list and the value of `p` is smaller than the new value. First `prev = p`, then `p = p->link` is written; this order guarantees that `prev` always stays exactly one step behind `p`. When the loop ends, the place of the new node is between `prev` and `p`.

After the loop, `fresh->link = p` is written first. This preserves the address of `p`, whether it is a real node or `NULL`. Then `prev` is examined. If `prev` is still `NULL`, the loop never ran. That means either the list is empty or the new value is smaller than or equal to the first node; in both cases the new node goes to the front and `*head` changes. If `prev` is not `NULL`, the new node is inserted in the middle or at the end: `prev->link = fresh`.

Example 4.1

The value 7 is to be inserted into the sorted list 2 → 5 → 9. The new node has been allocated with `malloc` and its value 7 written. Table 4.3 gives each test of the loop on a separate row.

Step	`prev`	`p`	Condition	Action
Start	NULL	&node(2)	—	—
1st test	NULL	&node(2)	2 < 7: true	`prev = p`, `p = p->link`
2nd test	&node(2)	&node(5)	5 < 7: true	`prev = p`, `p = p->link`
3rd test	&node(5)	&node(9)	9 < 7: false	The loop ends; the place is between 5 and 9.
`fresh->link = p`	&node(5)	&node(9)	—	7 points to 9.
`prev == NULL` ?	&node(5)	&node(9)	false	The head does not change.
`prev->link = fresh`	&node(5)	&node(9)	—	List: 2 → 5 → 7 → 9.

Table 4.3: Trace of inserting 7 into the list 2 → 5 → 9 with `sorted_insert`.

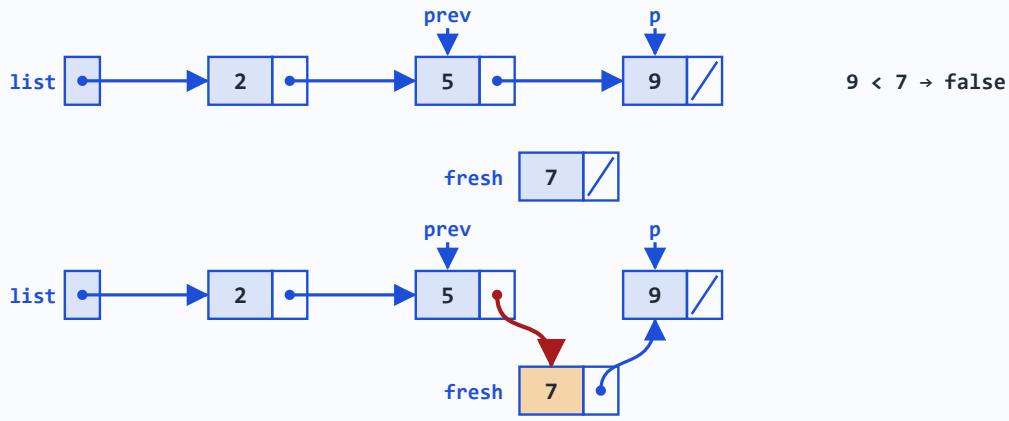


Figure 4.8: Two moments of sorted insertion. At the top the loop has stopped because 9 < 7 is false; `prev` is at 5, `p` is at 9 and the new node is not yet linked. At the bottom `fresh->link = p` and `prev->link = fresh` have been written: 2 → 5 → 7 → 9.

The same code handles the other cases correctly too (they were tried in the verification program). If 1 is inserted, the first test 2 < 1 is false, so the loop never runs; `prev` stays `NULL` and the new node goes to the front. If 10 is inserted, the loop passes all the nodes and stops when `p` becomes `NULL`; `fresh->link` becomes `NULL` and the new node is attached after the last node. If a value equal to an existing one is inserted, for example 5, the condition `5 < 5` is false, so the loop stops in front of the old 5 and the new 5 goes in before the old one. Had the condition been written `<=`, the new 5 would go in after the old one. When the relative order of equal values matters, this detail is decisive.

The time of sorted insertion is determined by the number of iterations of the loop. In the best case the new value is the smallest, the loop does not run and the time is $O(1)$. In the worst case the new value is the largest and the whole list is traversed: $\Theta(n)$. In a sorted array the right place can be found by binary search in $O(\log n)$, but the shifting that follows is again $O(n)$. In a list, finding the place is $O(n)$ and linking is $O(1)$. The total cost is linear in both structures, but its source is different.

Common mistake

Writing the loop condition in the order `p->data < fresh->data && p != NULL`. The `&&` operator is evaluated from left to right. When the end of the list is reached, `p` is `NULL` and the first condition tries to look inside `NULL`; the program crashes. In the right order (`p != NULL && …`), when the first condition is false the second is never evaluated. This behavior is called **short-circuit evaluation**, and every linked list loop relies on it.

4.5 Deletion, search and releasing the list

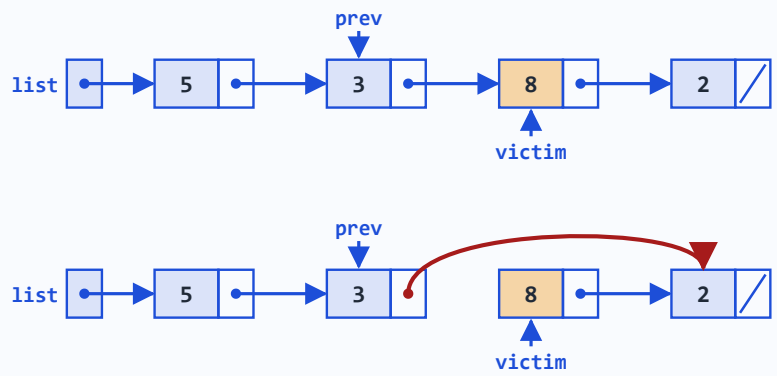
Deletion is the reverse of insertion, but with one extra step: after the node has been taken out of the chain, its memory must also be returned with `free`. This section first describes deleting the next node, then deleting by value. Search and positional access follow, and last come the recursive and iterative routines that release a whole list.

4.5.1 Deleting the next node

Deleting a node from a singly linked list means changing the link that points to it. That link is not in the node to be deleted but in the node before it. Deletion therefore needs the predecessor of the node to be deleted. The first deletion function in the slides carries this requirement directly into its interface: it takes not the node to be deleted but its predecessor.

```c
/* deletes the node after prev from the list
   and from memory */
void delete_after(node_ptr prev) {
    node_ptr victim = prev->link; /* keep the address */
    if (victim == NULL) return; /* nothing to delete */
    prev->link = victim->link;   /* repair the link */
    free(victim);                /* release the node */
}
```

The first line takes the address of the node to be deleted into the variable `victim`. If this address were not saved, after the link was changed there would be no way to reach the node and `free` could not be called. If `prev` is the last node there is no node after it, and the function returns without doing anything. Otherwise the line `prev->link = victim->link` jumps over the node to be deleted: `prev` now points to the successor of that node. After this line the node is out of the list but still sits in memory. The last line returns it to memory.



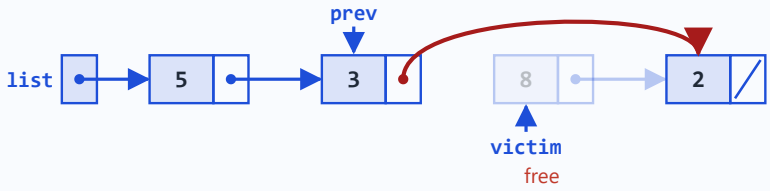


Figure 4.9: Deleting the node after 3 (the node 8) in the list 5 → 3 → 8 → 2. At the top, `victim = prev->link` has saved the address of 8. In the middle, `prev->link = victim->link` makes 3 point to 2; 8 is out of the chain. At the bottom, `free(victim)` has returned the memory of 8: 5 → 3 → 2.

The function performs two assignments and one `free`; its time is *O(1)*. But it has a limit: the head node cannot be deleted, because the head node has no predecessor. To delete the head node, the variable that holds the head of the list must be changed. There are two ways around this limit. The first is for the function to take the list by address and contain a separate branch for the head; the delete-by-value function is like this. The second is to put a header node, which is never deleted, at the front of the list. Then every real node has a predecessor and the special case disappears.

4.5.2 Deleting by value

In most applications the node to be deleted is known not by its address but by its value: "delete 1 from the list". In this case the node is first found and then deleted. During the search, as in sorted insertion, a trailing `prev` pointer is kept.

```
bool delete_value(node_ptr *head, int x) {
    node_ptr prev = NULL, p = *head;
    while (p != NULL && p->data != x) {
        prev = p;
        p = p->link;
    }
    if (p == NULL) return false;          /* x absent */
    if (prev == NULL) *head = p->link;  /* the head */
    else prev->link = p->link;          /* skip it */
    free(p);
    return true;
}
```

- The loop advances the two pointers together as long as `p` has not reached the end of the list and the value of `p` differs from `x`.
- If the loop ended with `p == NULL`, `x` is not in the list; the function returns `false` and the list does not change.
- If `prev == NULL`, the loop never ran, that is, `x` is in the first node. The head changes: `*head = p->link`. If the list has a single node, `p->link` is `NULL` and the list becomes empty.
- Otherwise `prev->link = p->link` jumps over `p`.

- In both branches `p` is no longer in the list and is released with `free(p)`. The link change is made before `free`, because `p->link` has to be read.

Example 4.2

The list is 3 → 8 → 1 → 6 and `delete_value(&list, 1)` is called. Table 4.4 gives every step. The loop runs twice and stops at the third test because `1 != 1` is false. At this moment `p` points to 1 and `prev` to 8. Since `prev` is not `NULL`, the head does not change; the link of 8 is turned to 6 and the node of 1 is released. The result is 3 → 8 → 6.

Step	`prev`	`p`	Condition / action
Start	NULL	&node(3)	—
1st test	NULL	&node(3)	3 != 1: true → `prev = p`, `p = p->link`
2nd test	&node(3)	&node(8)	8 != 1: true → `prev = p`, `p = p->link`
3rd test	&node(8)	&node(1)	1 != 1: false → the loop ends
`p == NULL` ?	&node(8)	&node(1)	false: `x` is in the list
`prev == NULL` ?	&node(8)	&node(1)	false: the head is not being deleted
`prev->link = p->link`	&node(8)	&node(1)	8 now points to 6; 1 has been skipped
`free(p)`	&node(8)	(dangling)	The node was returned; the list is 3 → 8 → 6, the result is `true`

Table 4.4: Trace of `delete_value(&list, 1)`; initially the list is 3 → 8 → 1 → 6.

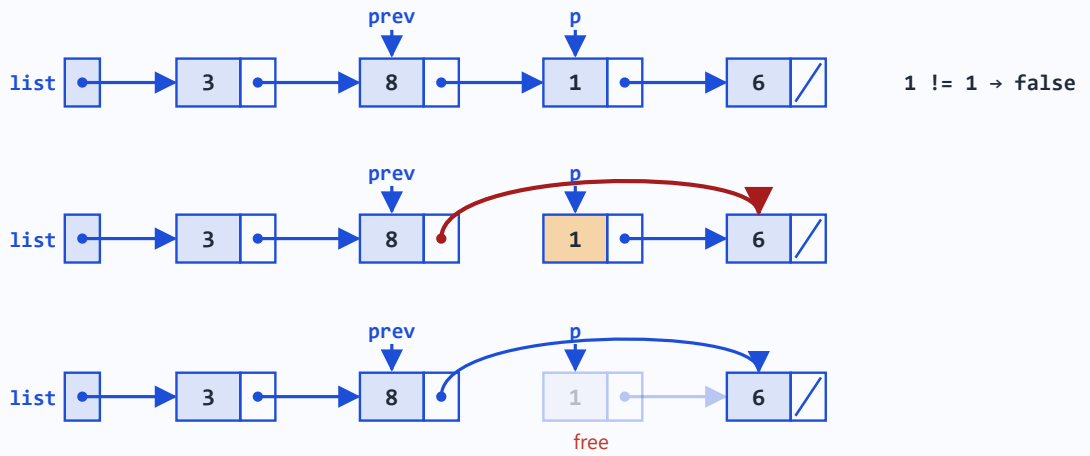


Figure 4.10: Three moments of the call `delete_value(&list, 1)`. At the top the loop has stopped because 1 != 1 is false; `prev` is at 8 and `p` is at 1. In the middle, `prev->link = p->link` makes 8 point to 6. At the bottom, after `free(p)`, the list is 3 → 8 → 6.

The cost of the function comes from the search. If `x` is in the k-th node (counting from 0), the loop runs k times; if `x` is not in the list, it runs n times. In the worst case the time is $\Theta(n)$. The link change and `free` after the search take constant time. In the verification program, from the same list first 1

was deleted, then 3 at the head, and finally 42, which is not in the list, was searched for. The three calls ran their loops 2, 0 and 2 times; the first two returned `true` and the last returned `false`. At the last call the list was 8 → 6, so searching for 42 took two iterations.

What happens if the call `free(p)` is forgotten? The node has left the list and the only variable that points to it is the local `p`. When the function returns, `p` disappears too. The node stays in memory but no pointer can reach it: this is the memory leak of the definition. Most leaks in list code come precisely from a forgotten `free` in a deletion function.

4.5.3 Search and positional access

Search is a form of traversal that ends early. It stops the moment the value is found and returns the address of the node. Positional access takes a given number of steps. The two functions in the slides do these two jobs:

```
node_ptr find(node_ptr p, int x) {
    while (p != NULL && p->data != x)
        p = p->link;
    return p;               /* NULL if not found */
}

node_ptr nth(node_ptr p, int i) {  /* 0-based */
    while (p != NULL && i-- > 0)
        p = p->link;
    return p;               /* NULL if i >= length */
}
```

`find` advances as long as `p` has not reached the end of the list and does not hold the sought value. The loop ends for two reasons. If `p` has stopped on a node, that node holds the sought value and its address is returned. If `p` has reached `NULL`, the value is not in the list and `NULL` is returned. The caller must therefore check for `NULL` before using the return value.

The order in the condition is decisive here too: first `p != NULL`, then `p->data != x`. If the order were reversed, when the value is absent `p` would become `NULL` after the last node and an attempt would be made to read `p->data`. Thanks to short-circuit evaluation, in the right order the second condition is never evaluated when the first is false.

The function `nth` finds the 0-based `i`-th node. The expression `i-- > 0` first compares `i` with 0 and then decrements `i`. If it is called with `i = 2`, the comparisons are 2 > 0, 1 > 0 and 0 > 0; since the first two are true, `p` advances two steps. If the list is shorter than `i`, `p` becomes `NULL` on the way and the function returns `NULL`. In the verification program, on the list 4 → 1 → 6 → 5, `nth(list, 0)` and `nth(list, 3)` returned 4 and 5 respectively, and `nth(list, 4)` returned `NULL`.

The costs of the two functions show plainly the weakest side of the linked list compared with the array. `find` takes $\Theta(n)$ in the worst case, and `nth(p, i)` takes $\Theta(i)$. In an array, `a[i]` is read in $O(1)$ with a single address computation. A list is therefore a poor choice in a program that frequently asks to "fetch the i-th element". For example, reading all elements with `nth(list, i)` for $i = 0, 1, \ldots, n - 1$ in a loop takes $0 + 1 + \ldots + (n - 1)$, that is, $\Theta(n^2)$ steps. The same job can be done in $\Theta(n)$ with a single traversal.

If the list is sorted, search can be improved a little. One stops the moment `p->data > x` is seen, because all later values are larger than `x` as well. When the value is absent, this halves the search on average. But the worst case is still $\Theta(n)$. Binary search, which is used on sorted arrays, is useless on a linked list. Binary search looks at the middle element at every step. In a list, even reaching the middle element requires walking half the list. The total work is $n/2 + n/4 + \ldots$, that is, again $\Theta(n)$, and the reduction in the number of comparisons gains nothing.

4.5.4 Releasing the list: an exam question

When a list is no longer needed, all of its nodes must be returned to memory. To do this each node is freed one by one. The slides treat this job through an old exam question. The question gives two routines and asks which is correct and what the other routine does to the list in the calling function. The first version is the faulty form that students often write:

```
void free_list( list_ptr *list ){
    node_ptr p = *list;
    if( *list ){
        p = *list->link;
        free(*list);
        free_list(&p);
}}
```

The second version is the correct one:

```
void free_list( list_ptr *list ){
    node_ptr p = *list;
    if( p ){
        *list = p->link;
        free(p);
        free_list(list);
}}
```

The first version has two separate problems. The first is a syntax error: the expression `*list->link` does not compile. The `->` operator has higher precedence than `*`, so the expression is read as `*(list->link)`. `list` points to a pointer, not to a node; `list->link` is meaningless. The compiler reports "member reference base type is not a structure or union" (this error was obtained in the verification program). The intended expression is `(*list)->link`.

Even if the syntax error is fixed, a second, deeper problem remains. The first version makes the recursive call with `&p`, that is, with the address of the *local* variable `p`. The second call therefore sees not the `list` of `main` but the local `p` of the first call. All nodes are still released, but `list` in `main` never changes. When the calls return, `list` still holds the address of the old first node. That node has been returned by now: `list` has become a dangling pointer. In the verification program the corrected first version was called with the list 4 → 9; on return the test `list == NULL` came out false and `list->data` could be read. This read is undefined behavior; that it returned the old value 4 in that run does not show that the program is correct, only that the bug stayed hidden.

The second version uses the same address `list` in every call. The line `*list = p->link` advances `list` in `main` directly to the next node; then the old head is released. In the last call `*list` is `NULL` and the recursion stops. On return, `list` in `main` is `NULL`; no dangling pointer is left.

This question also shows how easy it is for the person who writes code to trust their own code too much. The first version looks reasonable at first sight, and once the syntax error is fixed it gives the impression of working without trouble. The way to see the bug is to follow, one by one, which variable's address each pointer holds. The trace in the next subsection does this for the correct version.

4.5.5 Code trace: `free_list(&list)`

The slides trace the second version, tidied up and commented:

```
void free_list(node_ptr *list) {
    node_ptr p = *list;
    if (p) {
        *list = p->link;      /* advance the head first */
        free(p);              /* then free the old head */
        free_list(list);      /* continue, same address */
    }
}
```

The list is 4 → 9. Table 4.5 gives the state of the variables in each of the three calls. Every call has its own `p`, but the `list` parameter of all three calls holds the same address, the address of `list` in `main`.

Call	Line	`p`	`*list` (= `list`)	Explanation
1	`p = *list`	&node(4)	&node(4)	There is a node to free.
1	`*list = p->link`	&node(4)	&node(9)	`list` in `main` advanced.
1	`free(p)`	(dangling)	&node(9)	Node(4) was returned; `p` is no longer used.
2	`p = *list`	&node(9)	&node(9)	The second call, with the same address.
2	`*list = p->link`	&node(9)	NULL	`list` is now `NULL`.
2	`free(p)`	(dangling)	NULL	Node(9) was returned.

Call	Line	`p`	`*list` (= `list`)	Explanation
3	`p = *list`	NULL	NULL	`if (p)` is false: the base case, the call returns.
2, 1	return	—	NULL	No work is left; the calls return in turn.

Table 4.5: Trace of `free_list(&list)`; initially the list is 4 → 9.

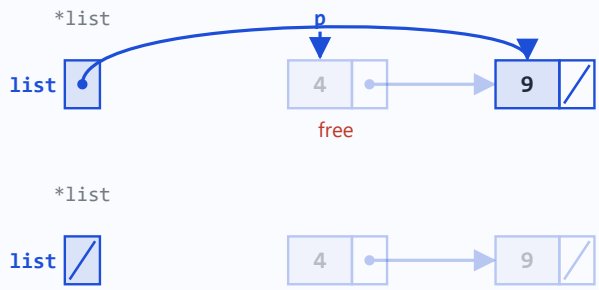


Figure 4.11: Two moments of the release. At the top, in the first call, `list` has advanced to 9 and node(4) has been returned with `free`. At the bottom, all nodes have been released and `list` in `main` has become `NULL`.

The order inside the body of the function is the same rule as in the deletion functions. First the address of the next node is moved to a safe place with `*list = p->link`, and then the old head is released. If the order were reversed, that is, if `free(p)` were written first, the following read of `p->link` would come from freed memory. After `free`, `p` must no longer be used; that is why it is marked “dangling” in the trace table.

For a list of n nodes the function is called $n + 1$ times: one call for each node, and a last call that sees `NULL`. Each call does constant work; the time is $\Theta(n)$. But recursion is not free. Each call takes room on the call stack for a frame that holds the return address, the parameter and the local `p`. At the deepest moment $n + 1$ frames are on the stack at once, and the extra memory is $\Theta(n)$.

4.5.6 Emptying with a loop

The call stack is a limited region of memory. Its size varies with the operating system and compiler settings, but it usually does not exceed a few megabytes. On a list of a million nodes the recursive version opens a million frames and may exceed the stack. This is called **stack overflow**. This case was tried in the verification program. When the code was compiled without optimization (`-O0`), a list of 100,000 nodes was released without trouble, while a list of 1,000,000 nodes crashed with a stack overflow error (`STATUS_STACK_OVERFLOW`). When the same code was compiled with `-O2`, a million nodes were released without trouble. The reason is that the recursive call is the last thing the function does. The compiler can turn such **tail recursion** into a loop. But the C standard does not guarantee this transformation; the safe way is to write the loop explicitly.

```
void clear_list(node_ptr *list) {
    while (*list != NULL) {
        node_ptr p = *list;
        *list = p->link;    /* advance first */
        free(p);            /* then free */
    }
}
```

The loop runs until the list is empty. In each iteration the address of the head node is taken into the local `p`. `*list = p->link` moves the list one node forward. Then `p` is freed. This is the body of the recursive version placed inside a loop. The time is again $\Theta(n)$, but the extra memory is only `p`: $O(1)$. When the loop ends, `*list` is `NULL`, so the caller's variable has been cleared as well.

The common rule of the recursive and iterative versions is this: the next address is taken before `free`; the link field of a freed node is never read. This rule was applied to all deletion and release functions in the verification program of these notes, and at the end of the program the calls to `malloc` and `free` were counted. 84 calls to `free` matched 84 calls to `malloc`, so no node leaked.

Common mistake

Writing the emptying loop as `while (*list != NULL) { free(*list); *list = (*list)->link; }`. The second line reads the link field of the node that has just been freed. In most runs the program gives the expected result, because the returned block has not been handed to another use immediately. But if another thread or the internal bookkeeping of `malloc` has written into that block, the loop jumps to a random address. Errors of this kind are caught by tools such as AddressSanitizer; they are hard to notice by eye.

4.6 Classic routines

Some problems about linked lists come up so often in exams, interviews and real code that their names have become classic. The four routines in this section share a common property: they traverse the list at most once, allocate no new nodes, and work with only a few pointers, that is, with $O(1)$ extra memory. The essence of all of them is walking several pointers together in a particular pattern.

4.6.1 Reversing a list

The slides introduce this routine with an old exam question: write a routine `reverse` that reverses the link fields of the list pointed to by the pointer `list`. The list 10 → 20 → 30 must become 30 → 20 → 10. The key phrase of the question is “link fields”. Copying the values into an array and writing them back in reverse order, or building a new list and moving the nodes by inserting them at the

front, would also give a result. But that is not what is asked. What is asked is to turn only the direction of the links, without allocating any node and without moving any value. There are no new nodes, no copies, no `malloc` and no extra memory; the pointers do all the work.

The idea is this: the list is traversed once from beginning to end, and the link of each node is turned so that it points to the previous node. But the moment the link of a node is turned, the way to the node after it is lost. The address of the next node must therefore be saved before turning. Also the value to be written into the turned link, that is, the address of the previous node, must be kept at hand. Three pointers are needed in all.

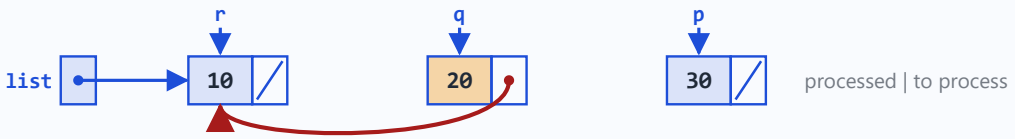


Figure 4.12: An intermediate state of reversal. The link between `r` and `q` has been turned (the processed part); everything from `p` on is still in its old direction (the part yet to be processed).

Figure 4.12 shows the roles of the three pointers. `r` is the pointer furthest behind; it holds the head of the reversed part, that is, the node the turned link will point to. `q` is in the middle and points to the node whose link will be turned in this iteration. `p` is furthest ahead and holds the head of the list not yet processed. In each iteration one link is turned and the three pointers advance one step together.

```
void reverse(node_ptr *list) {
    node_ptr r, p = *list, q = NULL;
    while (p) {
        r = q;          /* r trails q */
        q = p;          /* q trails p */
        p = p->link;    /* advance p */
        q->link = r;    /* reverse the link */
    }
    *list = q;          /* last node is the head */
}
```

Initially `p` points to the head of the list and `q` to `NULL`; the value of `r` does not matter yet. That `q` is `NULL` is no accident: since the first node will be the last node of the reversed list, its link must be `NULL`, and this value is written into it through `r` in the first iteration. Each iteration of the loop makes four assignments:

1. `r = q`: `r` moves to the node that was turned in the previous iteration. This node is the new target of the link to be turned in this iteration.
2. `q = p`: `q` moves to the node whose link will be turned in this iteration.

3. `p = p->link` : `p` advances one step. This assignment is made *before* the link is turned; otherwise the rest of the list would be lost.
4. `q->link = r` : the link of `q` is turned backwards, to `r` .

The loop ends when `p` becomes `NULL` . At this moment `q` points to the last node of the old list. This node is the first node of the reversed list and is made the head with `*list = q` . Because the head of the list changes, the function takes the list by address, as `node_ptr *list` .

4.6.2 Code trace: `reverse(&list)`

The list is 10 → 20 → 30. Table 4.6 gives the state at the end of each iteration. The last column of the table shows the links turned up to that moment.

Moment	`r`	`q`	`p`	Link turned in this iteration	State of the links
Start	?	NULL	&node(10)	—	10 → 20 → 30
Iteration 1	NULL	&node(10)	&node(20)	10 → NULL	NULL ← 10 · 20 → 30
Iteration 2	&node(10)	&node(20)	&node(30)	20 → 10	NULL ← 10 ← 20 · 30
Iteration 3	&node(20)	&node(30)	NULL	30 → 20	NULL ← 10 ← 20 ← 30
`*list = q`	&node(20)	&node(30)	NULL	—	List: 30 → 20 → 10

Table 4.6: Trace of `reverse` ; initially the list is 10 → 20 → 30. The values are the state at the end of each iteration.

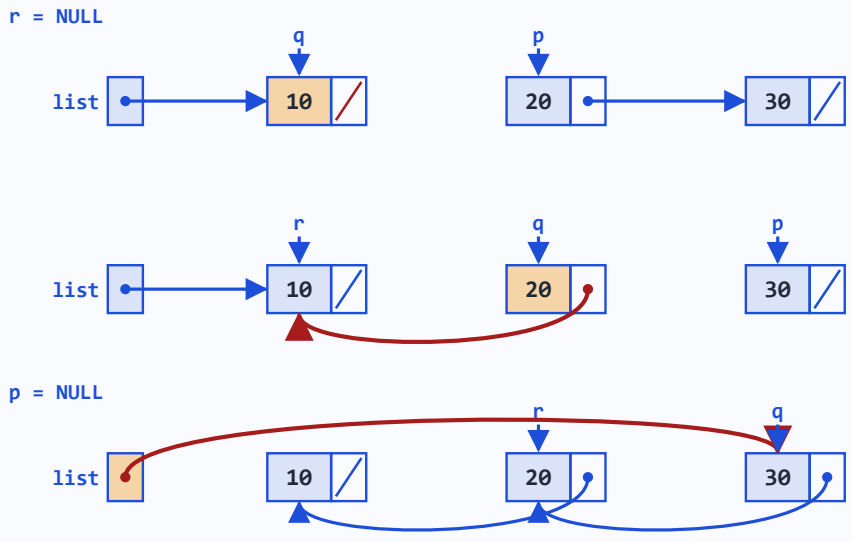


Figure 4.13: Three moments of reversal. At the top, at the end of the first iteration, node(10) points to `NULL` . In the middle, at the end of the second iteration, node(20) points to node(10); `p` is at 30. At the bottom, `*list = q` has made the old last node the new head: 30 → 20 → 10.

In the first iteration `r` takes the value `NULL`, which is written into the link of 10. So 10 becomes the last node of the reversed list. In the second iteration the link of 20 is turned to 10, and in the third the link of 30 to 20. In the middle of every iteration there is a moment when the list is split into two parts: the part turned backwards, reachable from `q`, and the part not yet turned, reachable from `p`. Since the head of each part is held in a pointer, no node is lost. In the verification program the function was applied twice to the same list and the list returned to its original state. On an empty list the loop never runs and the assignment `*list = q` writes `NULL`; the empty list stays empty.

On a list of *n* nodes the loop runs *n* times and each iteration makes four assignments. The time is *Θ(n)* and the extra memory is three pointers, that is, *O(1)*. The routine works **in place**: it allocates no new nodes and creates no copies. It therefore cannot fail for lack of memory; reversal can be done even when memory is full.

> **Common mistake**
>
> Swapping the lines `p = p->link` and `q->link = r` in the loop. At this point `q` and `p` point to the same node. If `q->link = r` is written first, `p->link` changes as well; the following `p = p->link` takes `p` not forward but backward, to `r`. In the first iteration `r` is `NULL`, so the loop ends immediately and the list consists only of the first node. All the remaining nodes are lost.

4.6.3 Concatenating two lists

Concatenating two lists consists of linking the last node of the first list to the first node of the second. No node is copied.

```
/* links L2 after the end of L1, returns the head */
node_ptr concat(node_ptr L1, node_ptr L2) {
    node_ptr p;
    if (L1 == NULL) return L2;
    if (L2 != NULL) {
        for (p = L1; p->link != NULL; p = p->link)
            ;                   /* last node of L1 */
        p->link = L2;
    }
    return L1;
}
```

If the first list is empty the result is the second list, and the function returns `L2`. If the second list is empty there is nothing to do. If neither is empty, the `for` loop takes `p` to the last node of `L1` and `p->link = L2` is written. The function returns the head of the result list. This design differs from the insert-at-end function in that it takes the list by value and delivers the new head through the return value. The caller assigns the new head to its own variable: `list = concat(L1, L2);`. Both designs are correct; passing by address needs less typing, while the return value gives a clearer interface.

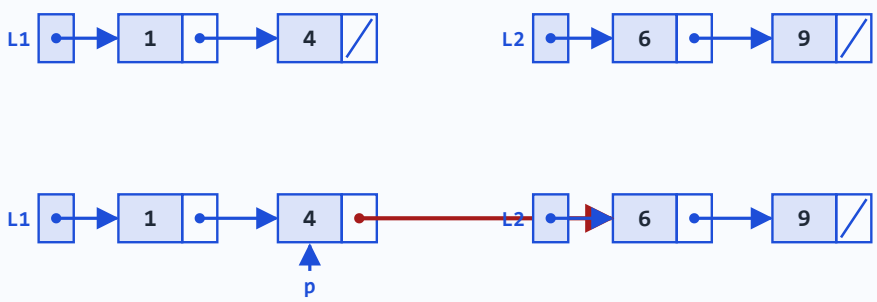


Figure 4.14: Concatenating the lists `L1` = 1 → 4 and `L2` = 6 → 9. At the top there are two separate lists. At the bottom `p` has found the last node of `L1` (4) and `p->link = L2` has been written: 1 → 4 → 6 → 9.

The cost comes from finding the last node of the first list: *Θ(|L1|)*. The length of the second list does not matter. Concatenating two arrays, on the other hand, requires copying the elements into a new array and takes *Θ(n + m)*. If the last node of the first list is kept in a `tail` pointer, concatenation drops to *O(1)*. In a circular list the job is again *O(1)*, because a circular list is held by its last node anyway.

Concatenation has a side effect: after the operation the nodes of `L2` belong jointly to both lists. The variable `L2` still points to 6, and a change made through `L2` changes the combined list as well. Care is needed especially when releasing. If, after the combined list has been released, `L2` is also released, the same nodes are freed a second time. Setting `L2` to `NULL` after concatenation prevents this error.

4.6.4 The middle node

The first way that comes to mind to find the middle node of a list takes two passes: first the length *n* is counted, then *n/2* steps are walked from the head. This is correct and takes *Θ(n)*. But the same job can be done in a single pass. For this, two pointers set out together from the head: the **slow pointer** takes one step per iteration and the **fast pointer** takes two. When the fast pointer reaches the end of the list, the slow pointer is exactly halfway.

```
node_ptr middle(node_ptr head) {
    node_ptr slow = head, fast = head;
    while (fast != NULL && fast->link != NULL) {
        slow = slow->link;          /* 1 step */
        fast = fast->link->link;    /* 2 steps */
    }
    return slow;
}
```

The loop condition guarantees that the fast pointer can take two steps. The test `fast != NULL` catches the case in which, in a list of even length, the fast pointer passes the last node and falls onto `NULL`. The test `fast->link != NULL` catches the case in which, in a list of odd length, the fast pointer

stops on the last node. The second condition also prevents the read `fast->link->link` from going through a `NULL` pointer.

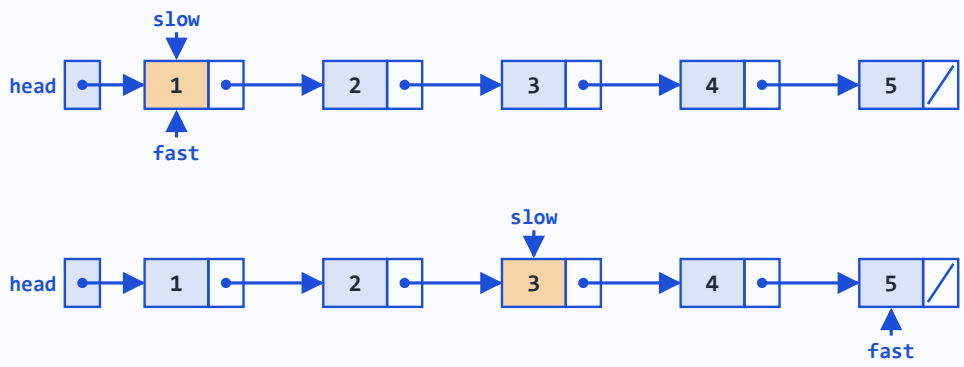


Figure 4.15: The middle node of the list 1 → 2 → 3 → 4 → 5. At the top both pointers are at the head. At the bottom `fast` has reached the last node (5); since `fast->link == NULL` the loop ends and `slow` points to the middle node (3).

In the five-node list 1 → 2 → 3 → 4 → 5 the loop runs twice. In the first iteration `slow` moves to 2 and `fast` to 3. In the second iteration `slow` moves to 3 and `fast` to 5. Since the link of 5 is `NULL`, the loop ends and the function returns 3. With even length there are two middles. On the list 1 → 2 → 3 → 4 the loop again runs twice: `slow` moves to 2 and 3, `fast` to 3 and `NULL`. The function returns the second middle, 3. If the first middle (2) is wanted, the condition `fast->link->link != NULL` is added; then the fast pointer stops one step earlier. All four of these values were obtained in the verification program.

On a list of n nodes the loop runs $\lfloor n/2 \rfloor$ times. The fast pointer takes about n steps in total and the slow pointer $n/2$; the total number of link reads is of the same order as in the two-pass solution. The gain is the single pass: the list is traversed once, and the length need not be known in advance. The time is $\Theta(n)$ and the extra memory is $O(1)$. The same “slow and fast pointer” idea gives a much stronger result in the next problem.

4.6.5 Floyd's cycle detection

In a correctly built linked list the link of the last node is `NULL`. A faulty link can break this arrangement. For example, if the link of the last node is mistakenly written to point to a node in the middle (`last->link = middle_node`), the list enters a **cycle**. In such a list the steps `p = p->link` never reach `NULL`. `print_list` prints forever, and a loop that counts the length never ends. Telling whether a list contains a cycle is called **cycle detection**.

The first solution that comes to mind is to keep the addresses of the visited nodes in a set. At each step one checks whether the new node is in the set; if it is, a cycle has been found. This solution works but needs $O(n)$ extra memory. The method attributed to Robert W. Floyd does the same job with $O(1)$

extra memory. The method is also known as “the tortoise and the hare”: the tortoise (`slow`) takes one step per iteration and the hare (`fast`) takes two.

```
bool has_cycle(node_ptr head) {
    node_ptr slow = head, fast = head;
    while (fast != NULL && fast->link != NULL) {
        slow = slow->link;         /* 1 step */
        fast = fast->link->link;   /* 2 steps */
        if (slow == fast) return true;
    }
    return false;
}
```

The skeleton of the code is the same as that of the middle-node function; the only difference is the test `slow == fast` at the end of each iteration. What is compared here is the addresses of the nodes, not their values. Two different nodes can hold the same value; but if two pointers hold the same address, they are on the same node. If the list has no cycle, the hare, being faster than the tortoise, reaches `NULL` before it, and the function returns `false` . If the list has a cycle, both pointers enter the cycle and go around it forever. Inside the cycle the hare gains one step on the tortoise in every iteration; so they must meet.

Theorem 4.1 (The tortoise and the hare meet)

Let there be μ steps from the head of the list to the first node of the cycle, and let the cycle consist of λ nodes. In `has_cycle` the two pointers meet at the same node within at most λ iterations after entering the cycle. The total number of iterations does not exceed $\mu + \lambda$, which equals the number n of distinct nodes in the list.

Proof. After t iterations the tortoise is t steps from the head and the hare is $2t$ steps away. As soon as $t \geq \mu$, both are inside the cycle. Inside the cycle, positions are measured modulo λ. Let d be the distance by which the hare is ahead of the tortoise, counted forward around the cycle; $0 \leq d < \lambda$. In each iteration the hare takes two steps and the tortoise one, so this distance decreases by 1. After d iterations the distance is 0 and the two pointers are on the same node. Since $d < \lambda$, the meeting happens at most $\lambda - 1$ iterations after the tortoise enters the cycle. The tortoise enters the cycle in iteration μ; the total number of iterations is therefore less than $\mu + \lambda$. If there is no cycle, the hare reaches `NULL` in at most $n/2$ iterations. In both cases the time is $O(n)$. ■

The important point in the proof is that the hare cannot “jump over” the tortoise. The hare takes two steps per iteration, but the distance between them decreases by exactly 1 per iteration. When the distance is 1 it becomes 0 one iteration later; when it is 2 it goes first to 1 and then to 0. The distance never becomes negative. If the hare took three steps, the distance would decrease by 2 at a time, and with an odd distance the meeting could be missed.

Example 4.3

The list is 1 → 2 → 3 → 4 → 5 and the link of 5 mistakenly points to 3. The cycle is 3 → 4 → 5 → 3; $\mu = 2$, $\lambda = 3$. Both pointers start at 1. Table 4.7 gives the end of each iteration. In the third iteration both pointers are at 4 and the function returns `true`. The number of iterations is 3, below the bound $\mu + \lambda = 5$ of the theorem.

Iteration	`slow`	`fast`	Loop condition	`slow == fast`
Start	1	1	—	—
1	2	3	1 and the successor of 1 are not NULL	false
2	3	5	3 and the successor of 3 are not NULL	false
3	4	4 (5 → 3 → 4)	5 and the successor of 5 are not NULL	true: `return true`

Table 4.7: Trace of `has_cycle` on the list 1 → 2 → 3 → 4 → 5 → (3) with a cycle.

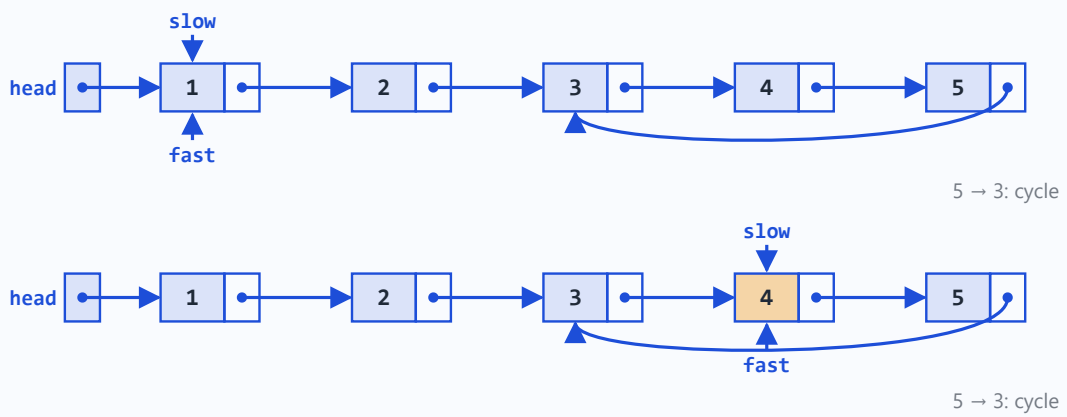


Figure 4.16: The hare and the tortoise in a list with a cycle. At the top both are at the head; the link from 5 back to 3 forms the cycle. At the bottom, at the end of the third iteration, the hare has gone around the cycle and caught up with the tortoise; both are at 4.

Had the same list had no cycle, that is, had the link of 5 been `NULL`, the hare would have reached 3 in the first iteration and 5 in the second. At the third test `fast->link == NULL` would end the loop and the function would return `false`. The verification program ran both cases and produced exactly the positions in the table.

4.6.6 The start of the cycle

Knowing that a cycle exists is often not enough; to repair the fault one also needs to know at which node the cycle begins. Floyd's method gives this too with *O(1)* extra memory. After the meeting, one of the pointers is moved back to the head of the list. Then both are advanced *one* step per iteration. The node where they meet again is the first node of the cycle. In the verification program the method was written as follows:

```
node_ptr cycle_start(node_ptr head) {
    node_ptr slow = head, fast = head;
    while (fast != NULL && fast->link != NULL) {
        slow = slow->link;
        fast = fast->link->link;
        if (slow == fast) {          /* a cycle exists */
            slow = head;               /* one goes back */
            while (slow != fast) {     /* one step each */
                slow = slow->link;
                fast = fast->link;
            }
            return slow;               /* first cycle node */
        }
    }
    return NULL;                       /* no cycle */
}
```

Theorem 4.2 (The start of the cycle)

If, after the meeting, one pointer is moved to the head and both pointers are advanced one step at a time, they meet exactly μ steps later at the first node of the cycle.

Proof. Let the meeting happen in iteration t. At that moment the tortoise is t steps from the head, the hare is $2t$ steps away, and both are on the same node. Since the two positions denote the same node, their difference t is a multiple of the cycle length λ: in the extra t steps it took, the hare has gone around the cycle a whole number of times. Now the pointer that starts at the head reaches the first node of the cycle after μ steps. The pointer that starts at the meeting point is, after μ steps, $t + \mu$ steps from the head. The first node of the cycle is μ steps from the head, and since t is a multiple of λ, $t + \mu$ steps also lands on the same node. So the two pointers meet at the start of the cycle at step μ. They cannot meet earlier, because before step μ the pointer coming from the head is outside the cycle, while the other pointer is always inside it. ■

In the example the meeting happened in iteration 3, at 4. The pointer from the head follows the path $1 \to 2 \to 3$, and the pointer from the meeting point follows $4 \to 5 \to 3$. Both reach 3 in two steps ($\mu = 2$), and 3 is found as the start of the cycle. Finding the length of the cycle is even easier: a pointer starting at the meeting node is walked, counting, until it returns to the same node. This is one of the exercises at the end of the chapter.

Floyd's method is used outside linked lists as well. Any sequence in which each value is computed from the previous one by the same rule, for example the output of a random number generator, eventually enters a cycle. The method finds the start and the length of this cycle without storing the elements of the sequence. Knuth attributes the method to Floyd in an exercise about random number generators (3.1-6) in the second volume of *The Art of Computer Programming*.

4.7 The circular list and the header node

The singly linked list has two recurring annoyances. Insertion at the end and concatenation must search for the last node, and the insertion and deletion code contains a separate branch for the head node. Two small changes in this section remove these annoyances. In a circular list the last node points to the first node and the list is held by its last node. In a header node scheme, a node that carries no data and is never deleted is placed at the front of the list.

4.7.1 The circular list

Definition 4.4

A **circular linked list** is a linked list in which the link field of the last node points to the first node instead of `NULL`. The list is usually held by a pointer `last` that points to the last node. In an empty circular list `last` is `NULL`; in a list with one node, the node points to itself.

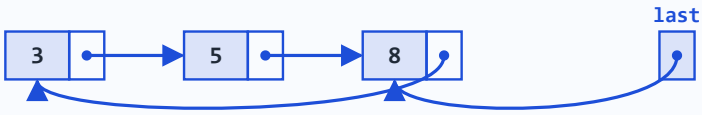


Figure 4.17: The circular list 3 → 5 → 8. The link of the last node (8) returns to the first node (3). The list is held by the pointer `last`; the first node is `last->link`.

Holding the end of the list instead of its head looks odd at first. But whoever knows the last node also knows the first: the first node is `last->link` and is found in one step. Whoever knows the head can find the end only by walking the whole list. The pointer `last` thus makes both ends accessible in *O(1)*. That is why, in a circular list, insertion both at the front and at the end takes constant time.

In a circular list no link is `NULL`. This changes the stopping condition of traversal. A loop written with the condition `p != NULL` never ends on a circular list. Traversal stops when it returns to the starting node. Since the starting node would satisfy the condition already in the very first iteration, the condition is checked at the end of the loop; this is why `do-while` is used. The length function in the slides shows this pattern:

```c
int c_length(node_ptr last) {
    int n = 0;
    node_ptr p = last;
    if (last == NULL) return 0;   /* empty list */
    do {
        n++;
        p = p->link;
    } while (p != last);          /* full circle */
    return n;
}
```

The empty list is checked separately, because the body of a `do-while` runs at least once and `p->link` cannot be read on an empty list. If the list is not empty, `p` starts at the last node. The body first increments the counter and then advances `p`. The condition keeps the loop going until `p` is back at the last node. On the list 3 → 5 → 8, `p` moves to 3, 5 and 8 in turn; after the third iteration `p == last` and the function returns 3. The time is $\Theta(n)$. Had the same loop been written as `while (p != last)`, the body would never run, because `p` is initially already equal to `last`, and the result would be 0.

4.7.2 Insertion into a circular list

In a circular list, insertion at the front means putting the new node between the last node and the first node:

```c
void c_push_front(node_ptr *last, node_ptr fresh) {
    if (*last == NULL) {          /* empty list */
        *last = fresh;
        fresh->link = fresh;      /* points to itself */
    } else {
        fresh->link = (*last)->link;  /* old first node */
        (*last)->link = fresh;
    }
}
```

If the list is empty, the new node becomes the only node. `last` points to it and the link of the node returns to itself: a circular list with one node. If the list is not empty, two links are written. The line `fresh->link = (*last)->link` links the new node to the old first node. `(*last)->link = fresh` links the last node to the new node; the new node is now the first node. Here too the order obeys the "save first, then link" rule. The parentheses in `(*last)->link` are mandatory; `*last->link` is read as `*(last->link)` and does not compile.

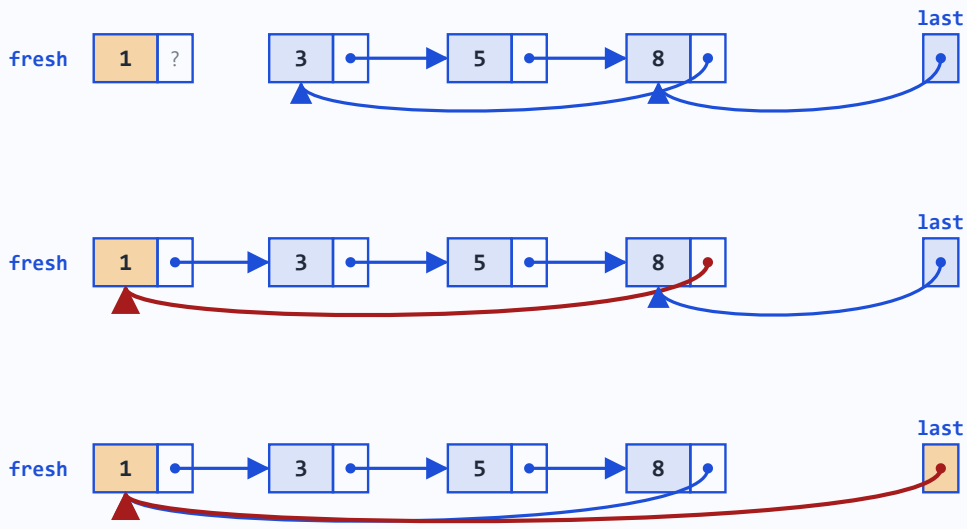


Figure 4.18: Inserting 1 into the circular list 3 → 5 → 8. At the top the new node is ready. In the middle, `fresh->link = last->link` and `last->link = fresh` have inserted 1 at the front: 1 → 3 → 5 → 8. At the bottom `*last = fresh` has also been written, so that 1 is inserted at the end: 3 → 5 → 8 → 1.

No separate function is needed for insertion at the end. In a circular list, "first" and "last" depend only on where the pointer `last` stands. If, after the new node has been inserted at the front, `last` is advanced by one node, that is, `*last = fresh` is written, the same node has been inserted at the end: `c_push_front(last, fresh); *last = fresh;`. Insertion at the front and at the end is the same link operation; the difference lies only in the pointer `last`. Both are *O(1)*. In the verification program, insertion at the front into the list 3 → 5 → 8 gave 1 → 3 → 5 → 8 and insertion at the end gave 3 → 5 → 8 → 1.

The circular list is a natural structure for data that is traversed again and again in order. The round-robin job scheduler of an operating system keeps the ready processes in a circular list; in each time slice it moves on to the next process, and when it reaches the end of the list it wraps around to the beginning by itself. In a multiplayer game, whose turn it is can be kept in the same way.

4.7.3 The header node

All the insertion and deletion code in the previous sections made the same distinction: if the change is at the head node, `*head` changes; otherwise the link of the previous node changes. This distinction needs an `if (prev == NULL)` branch in every function and passing by address in every call. Its origin is simple: the head node has no predecessor. The header node removes this deficiency.

Definition 4.5

A **header node** is a node that stands at the front of the list, carries no data and is never deleted. The list variable always points to the header node. The empty list consists of the header node alone. Because it makes boundary checks unnecessary, such a node is also called a **sentinel** node in some sources.



non-empty list: H → 3 → 8 empty list: H only

Figure 4.19: A list with a header node. On the left, a non-empty list: `list` points to the header (H) and the real nodes come after H (H → 3 → 8). On the right, the empty list: only H exists.

The gain of the header node is this: every real node has a predecessor. The predecessor of the first real node is the header. The list variable points to the header, and since the header is never deleted, this variable never changes. There is no need to give the functions the address of the list; the address of the header is enough. The sorted insertion function therefore becomes much shorter:

```
/* sorted insertion with a header: no special case */
void h_sorted_insert(node_ptr header, node_ptr fresh) {
    node_ptr prev = header;         /* never NULL */
    while (prev->link != NULL &&
           prev->link->data < fresh->data)
        prev = prev->link;
    fresh->link = prev->link;
    prev->link = fresh;
}
```

`prev` starts at the header and is never `NULL`. The loop advances as long as the successor of `prev` exists and is smaller than the new value. A single pointer is enough here, because the node being examined is always `prev->link`. When the loop ends, the new node is inserted after `prev`. Insertion at the front, in the middle, at the end and into the empty list: all four are done with the same two lines. The special case `if (prev == NULL)` has disappeared. In the verification program 8, 3, 5, 1 and 9 were inserted in this order into an empty list with a header, and the result was 1 → 3 → 5 → 8 → 9. In this sequence 3 and 1 went to the front and 9 to the end.

The price of the header node is the memory of a single node. In some applications the data field of the header does not go to waste; it is used to hold summary information such as the length of the list. In traversal and printing code, however, the header must be skipped: the real nodes start at `list->link`. The header node shows its greatest benefit when it is used together with doubly linked and circular lists.

4.8 The doubly linked list

In a singly linked list it is not possible to go backwards from a node. To delete a node or to insert in front of it, its predecessor is needed, and the predecessor can only be found by searching from the head. If a second link that holds the address of the previous node is added to each node, this restriction disappears.

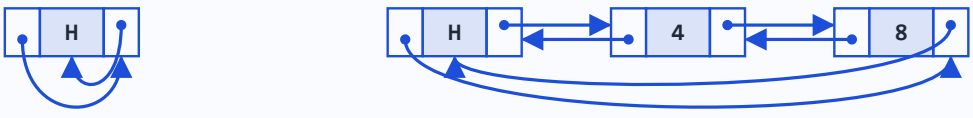
4.8.1 Structure and invariant

Definition 4.6

A **doubly linked list** is a linked list in which each node holds the address of both the next and the previous node. In the implementation in the slides, the field that points to the previous node is called `llink` and the field that points to the next node is called `rlink`.

```
typedef struct dnode *dnode_ptr;
struct dnode {
    dnode_ptr llink;   /* previous */
    int data;
    dnode_ptr rlink;   /* next */
};
/* invariant: p == p->llink->rlink == p->rlink->llink */
```

The slides build the doubly linked list as circular and with a header node. The `rlink` of the header node points to the first real node and its `llink` to the last real node. The `rlink` of the last real node and the `llink` of the first real node return to the header. So the list has no `NULL` link at all. In the empty list both links of the header point to itself.



empty: H points to itself

H ⇄ 4 ⇄ 8 ⇄ H

Figure 4.20: A circular doubly linked list with a header. On the left the empty list: both links of H point to itself. On the right H ⇄ 4 ⇄ 8 ⇄ H; the `rlink` of each node points to the next, the `llink` to the previous, and the two ends return to the header.

In this structure an equality holds for every node. The right neighbor of a node's left neighbor is the node itself; the left neighbor of its right neighbor is also itself. In short, `p == p->llink->rlink == p->rlink->llink`. This equality is called the **invariant** of the structure: it must hold for all nodes before and after every operation. In the circular structure with a header, the equality holds for the header and for the end nodes alike; no node is special. When writing insertion or deletion code, the question to ask is this: when the operation is over, is the invariant restored at all nodes? The verification program of these notes checked this equality for all nodes after every operation.

The doubly linked list has three important gains. The list can be traversed in both directions: `p = p->rlink` goes forward and `p = p->llink` goes backward. A node whose address is known is deleted in *O(1)* without searching for its predecessor. Insertion both before and after a node is done in *O(1)*. There are also three costs. Each node carries two pointers; on a 64-bit system this is an extra 8 bytes per node. Each insertion and deletion updates four links. With more links, it also becomes easier to make a link error.

4.8.2 Insertion

The insertion function in the slides inserts a new node to the right of the node `p`:

```
/* inserts the new node to the right of p */
void dinsert(dnode_ptr p, dnode_ptr fresh) {
    fresh->llink = p;
    fresh->rlink = p->rlink;
    p->rlink->llink = fresh;
    p->rlink = fresh;
}
```

Each of the four lines writes one link. The first two lines only prepare the new node: `fresh->llink = p` links the left of the new node to `p`, and `fresh->rlink = p->rlink` links the right of the new node to the current right neighbor of `p`. After these two lines the list itself has not changed at all; someone traversing from the front or from the back does not see the new node. The third line, `p->rlink->llink = fresh`, turns the `llink` of the node to the right of `p` toward the new node. The fourth line, `p->rlink = fresh`, turns the `rlink` of `p` toward the new node.

The order of the third and fourth lines cannot be changed. The third line reaches the node on the right of `p` through `p->rlink`. If the fourth line ran first, `p->rlink` would now be the new node, and `p->rlink->llink = fresh` would turn the left of the new node toward itself. The `llink` of the real right neighbor would still point to `p`; the invariant would be broken. The first two lines, on the other hand, can be written in either order, because they write only to the new node.

Example 4.4

The list is H ⇄ 4 ⇄ 8 ⇄ H. A new node with value 6 is to be inserted to the right of 4; `p` is

node(4). Table 4.8 gives the link that changes after each line.

Line	Link that changes	Explanation
`fresh->llink = p`	6.llink → 4	The left link of the new node points to 4.
`fresh->rlink = p->rlink`	6.rlink → 8	Its right link points to 8. The list has not changed yet.
`p->rlink->llink = fresh`	8.llink → 6	Since `p->rlink` is still 8, the left of 8 is written.
`p->rlink = fresh`	4.rlink → 6	The last link; the list is H ⇄ 4 ⇄ 6 ⇄ 8 ⇄ H.

Table 4.8: Trace of `dinsert(p, fresh)` ; `p` = node(4), `fresh` = node(6), initially H ⇄ 4 ⇄ 8 ⇄ H.

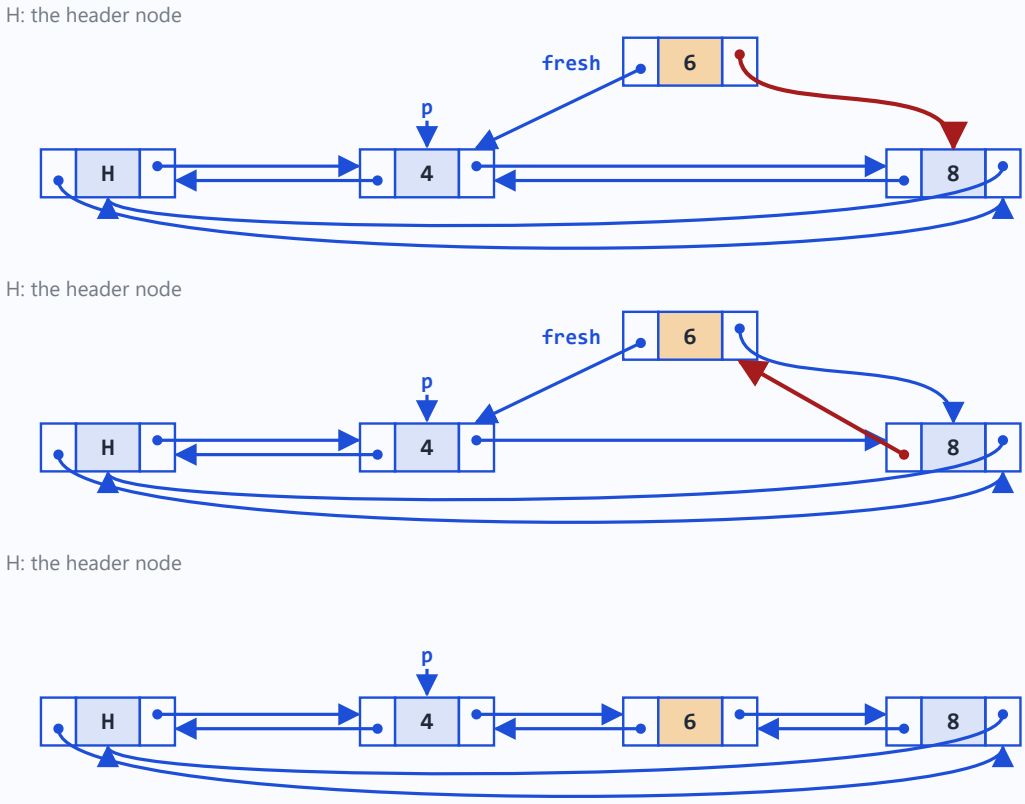


Figure 4.21: Three moments of doubly linked insertion. At the top the new node has been prepared: its left points to 4 and its right to 8. In the middle, `p->rlink->llink = fresh` has turned the left of 8 to 6. At the bottom, `p->rlink = fresh` has completed the insertion: H ⇄ 4 ⇄ 6 ⇄ 8 ⇄ H.

The function has no `if` at all. Whether `p` is the header (insertion at the front), the last node (insertion at the end) or the list is empty, the same four lines work correctly. In the empty list `p` is the header and `p->rlink` is the header too. Both links of the new node point to the header, and both links of the header point to the new node. This generality is the common gain of the header node and the circular structure. The time is four assignments, that is, *O(1)*.

4.8.3 Deletion

In a doubly linked list the predecessor of the node to be deleted is already at hand: `p->llink`. Deletion therefore consists of linking the node's two neighbors to each other and releasing the node:

```
void ddelete(dnode_ptr header, dnode_ptr p) {
    if (p == header) {       /* the header is never deleted */
        fprintf(stderr, "Cannot delete the header\n");
        return;
    }
    p->llink->rlink = p->rlink;
    p->rlink->llink = p->llink;
    free(p);
}
```

The header node must never be deleted; if it were, the list variable would turn into a dangling pointer. The function checks for this case and returns with an error message. Then two links are written. The line `p->llink->rlink = p->rlink` turns the `rlink` of the left neighbor toward the right neighbor. The line `p->rlink->llink = p->llink` turns the `llink` of the right neighbor toward the left neighbor. The order of these two lines does not matter, because both read the links of `p` itself and the links of `p` do not change. Finally the node is released.

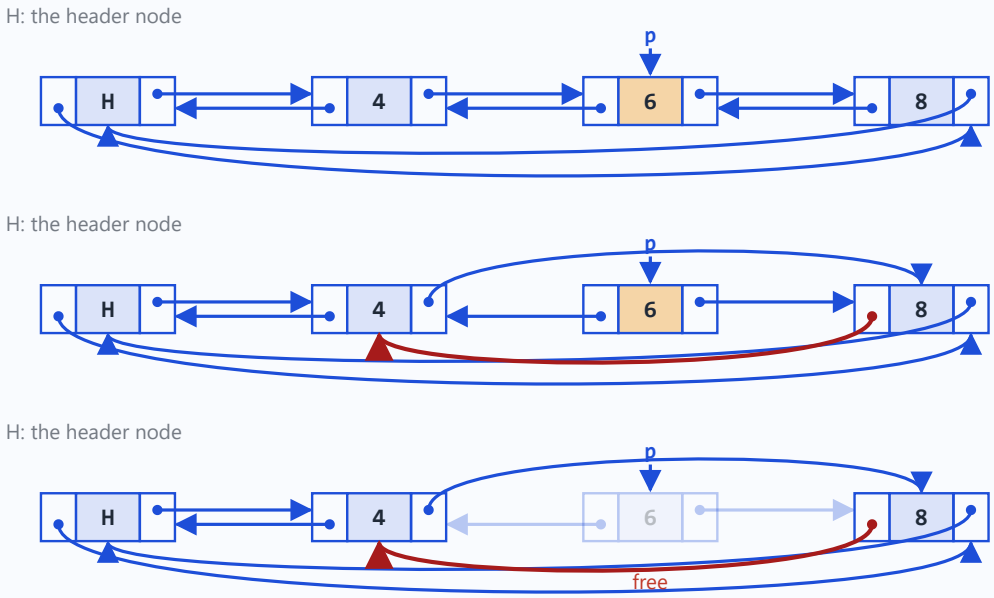


Figure 4.22: Deleting 6 from the list H ⇄ 4 ⇄ 6 ⇄ 8 ⇄ H. At the top `p` points to node(6); its predecessor is at hand through `p->llink`. In the middle the two neighbors have been linked to each other: the `rlink` of 4 is 8 and the `llink` of 8 is 4. At the bottom the node has been returned with `free(p)`.

There is no search in deletion; the time is *O(1)*. In a singly linked list the same operation would take *O(n)* to find the predecessor. Thanks to the header node, no separate code is needed for the first and last nodes either. When the first real node is deleted, `p->llink` is the header and the `rlink` of the header is updated. When the last real node is deleted, `p->rlink` is the header and the `llink` of the header is updated. In the verification program, from the list H ⇄ 4 ⇄ 6 ⇄ 8 ⇄ H first 6 was deleted, then 4 and 8; the invariant held at every step and in the end both links of the header pointed to itself. An attempt to delete the header was rejected with the message “Cannot delete the header”.

Common mistake

Updating the links in only one direction in a doubly linked list. For example, forgetting the line `p->rlink->llink = fresh` in insertion. Forward traversal works correctly and the bug goes unnoticed. But in backward traversal, going left from 8 skips 6 and arrives directly at 4. In a structure that shows different lists in the two directions, the next deletion scrambles the links completely. A small test function that checks the invariant (`p == p->llink->rlink == p->rlink->llink`) at all nodes after every operation catches this bug immediately.

4.9 The available-node list

Some programs keep inserting and deleting nodes throughout their run. For example, a polynomial-processing program creates new term nodes in every addition and multiplication, and it also releases the nodes of intermediate results immediately. Calling `malloc` at every insertion and `free` at every deletion is correct but not cheap. These two functions have to look for a free block of suitable size in memory and update their own bookkeeping. The solution in the book by Horowitz, Sahni and Anderson-Freed is to collect deleted nodes on the side instead of releasing them, and to take from there first when a new node is needed.

4.9.1 `get_node` and `ret_node`

Definition 4.7

An **available-space list** is a linked list to which deleted nodes are added instead of being freed, and which is consulted first when a new node is needed. The head of the list is traditionally kept in a global variable called `avail` .

```
node_ptr avail = NULL;        /* available nodes */

node_ptr get_node(void) {
    node_ptr p;
    if (avail != NULL) {         /* take from the list first */
        p = avail;
        avail = avail->link;
    } else {                     /* else ask the system */
        p = malloc(sizeof *p);
        if (p == NULL) exit(1);
    }
    return p;
}

void ret_node(node_ptr p) {      /* instead of free */
    p->link = avail;
    avail = p;
}
```

`get_node` takes the place of `malloc`. If `avail` is not empty, the first node of the list is taken and `avail` is advanced by one node. In this case `malloc` is not called at all. If `avail` is empty, a node is requested from the system; if there is no memory, the program ends. `ret_node` takes the place of `free`: the given node is added to the front of `avail`. Both functions perform a constant number of assignments and take *O(1)*.

On careful inspection, `avail` turns out to be a stack. `ret_node` adds at the front of the list and `get_node` takes from the front of the list. The node returned last is reused first: last in, first out. That is, `get_node` is the same as the pop operation of a linked stack and `ret_node` the same as push. Since insertion at the front and deletion from the front are the two operations that take constant time in a linked list, this choice is no coincidence.

The method has a limitation: the nodes in `avail` do not return to heap memory until the program ends. If a program uses a large number of nodes at one moment and then deletes most of them, the deleted nodes wait in `avail` and no other data structure can use that memory. Also, `avail` is useful only for a single node type. A program that uses nodes of different sizes must keep a separate `avail` for each type. The method is therefore suitable for cases in which there is a single node type and insertions and deletions are very frequent.

4.9.2 Step by step

Figure 4.23 starts from a state in which two free nodes are waiting in the `avail` chain. On the right there is a node in use (9).

1. `get_node()` is called. Since `avail` is not empty, `p = avail` is executed; `malloc` is not called.
2. With `avail = avail->link` the head of the chain advances. The node taken now belongs to the caller; one node is left in `avail`.

3. Later the node of 9 will be deleted. Instead of `free`, `ret_node(deleted)` is called and `deleted->link = avail` is written first.
4. With `avail = deleted` the node moves to the head of the chain and is ready for reuse.

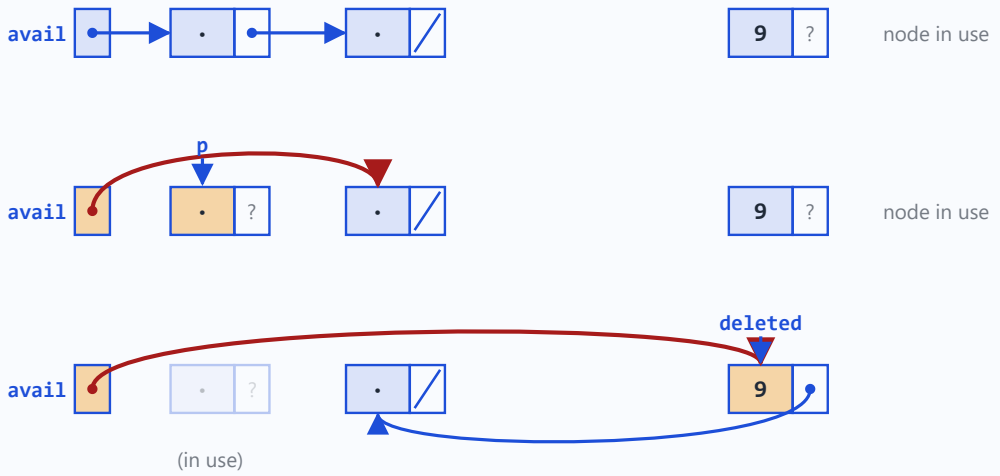


Figure 4.23: Taking and returning nodes with the `avail` list. At the top there are two free nodes in the `avail` chain. In the middle `get_node` has handed the first node to `p` and `avail` has advanced. At the bottom the deleted node has been added to the head of the chain with `ret_node`.

In the verification program this behavior was observed with counters. `get_node` calls made while nodes were waiting in `avail` did not increase the `malloc` counter and returned the node that had been returned last. While waiting in the chain, nodes may carry their old values in their data fields. `get_node`, like `malloc`, does not initialize the fields; that is the caller's job.

Common mistake

Continuing to use a node after returning it with `ret_node`. Unlike `free`, this error is caught by no tool, because the memory of the node is actually valid. But the link field of the node is now part of the `avail` chain. Every value written to it corrupts `avail`, and the next `get_node` call may hand out a random address as a node.

4.9.3 Returning a circular list in one move

Returning all the nodes of a list to `avail` requires, in a plain singly linked list, one `ret_node` per node, or at least a walk to find the last node, that is, $\Theta(n)$. In a circular list the same job is $O(1)$ regardless of length. The routine called `cerase` in Horowitz's book opens the ring at a single point and puts the whole list in front of `avail`:

```
/* return a circular list to avail in O(1) */
void cerase(node_ptr *last) {
    node_ptr first;
    if (*last != NULL) {
        first = (*last)->link;   /* first node */
        (*last)->link = avail;   /* open the ring */
        avail = first;
        *last = NULL;
    }
}
```

If the list is not empty, the address of the first node is first taken into `first` (`(*last)->link`). Then the link of the last node is turned to `avail`. This assignment opens the ring: the last node now points not to the first node but to the head of the `avail` chain. At this moment someone starting at `first` and moving forward passes through all the nodes of the list and reaches the old nodes of `avail`. The assignment `avail = first` makes this whole chain the new `avail`. Finally the caller's variable `last` is set to `NULL`; the list is now empty and no dangling pointer is left.

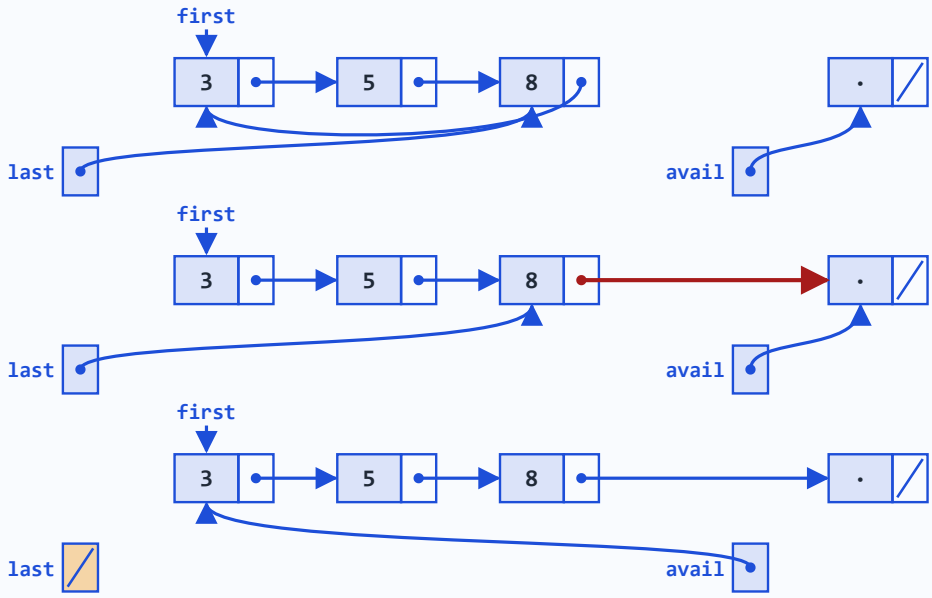


Figure 4.24: Returning the circular list 3 → 5 → 8 to `avail`. At the top `last` = 8, `first` = 3 and there is one node in `avail`. In the middle `last->link = avail` has opened the ring; 8 now points to the old head of `avail`. At the bottom `avail = first` and `*last = NULL` have put the whole list in front of `avail`.

The routine makes only four assignments and contains no loop; the time is the same whether the list has three nodes or three million. What makes this possible is that a circular list gives both of its ends in *O(1)*: `last` points to the last node and `last->link` to the first. In a plain singly linked list, finding the last node for the same job would take *Θ(n)*. In the verification program three nodes were taken

from an `avail` chain of eight nodes and the circular list 3 → 5 → 8 was built with them; five nodes remained in `avail`. After the `cerase` call, `avail` started with 3, 5, 8 and its length rose to eight again.

4.10 Complexity summary and measurement

Table 4.9 gives the costs of the operations of this chapter for four structures side by side. Every value in the table has been justified in the relevant section. The singly linked list is held by its head pointer alone, the circular list by the pointer `last`, and the doubly linked list by its header node.

Operation	Array	Singly linked (head)	Circular (last)	Doubly linked (header)
Access the *i*-th element	*O(1)*	*O(i)*	*O(i)*	*O(i)*
Insert at the front	*O(n)*	*O(1)*	*O(1)*	*O(1)*
Insert at the end	*O(1)**	*O(n)*	*O(1)*	*O(1)*
Insert after a known node	*O(n)*	*O(1)*	*O(1)*	*O(1)*
Delete a known node	*O(n)*	*O(n)*†	*O(n)*†	*O(1)*
Search for a value	*O(n)*	*O(n)*	*O(n)*	*O(n)*
Concatenation	*O(n + m)*	*O(n)*	*O(1)*	*O(1)*
Extra memory per node	—	1 pointer	1 pointer	2 pointers

Table 4.9: Operation costs for arrays and the kinds of linked list (*n* and *m* are list lengths).

* If the array's capacity is full, a resize is needed first; that step is *O(n)*. † Finding the predecessor of the node to be deleted requires a traversal; if the predecessor is known, deletion is *O(1)*. Reversal, the middle node and Floyd's cycle detection need *O(n)* time and *O(1)* extra memory in every kind of list.

To see what the table row “insertion at the end of a singly linked list is *O(n)*” means in practice, a small measurement was made. The measurement program inserted *n* elements in three ways: into an array, each time at the front (shifting the earlier ones with `memmove`), into a list with `push_front`, and into a list with `push_back`. The program was compiled with `-O2` and run several times. Table 4.10 gives one of the runs. The times vary with the machine and the run; what matters is how the numbers grow.

n	Array, at the front (shifting)	List, `push_front`	List, `push_back`
10,000	2.4	0.42	232
20,000	18.8	0.72	768
40,000	57.5	1.33	3,039

Table 4.10: Time to insert *n* elements (milliseconds). The times vary with the machine.

Insertion at the front of a list does constant work per element; the total time is $\Theta(n)$ and roughly doubles when n doubles. The other two columns grow quadratically: when n doubles, the time roughly quadruples. In the array every insertion shifts all the elements so far; in the list every `push_back` walks all the nodes so far. Both do $n(n-1)/2$ units of work in total.

The surprising side of the measurement is the gap between the two quadratic operations. Although they take the same number of steps, insertion at the end of the list is about fifty times slower than insertion at the front of the array. The reason is the way memory is accessed. `memmove` moves consecutive bytes in large blocks and uses the processor's cache very efficiently. A list walk, on the other hand, jumps to another place in memory at every step; the address of the next node is known only after the current node has been read. This is why, even when the asymptotic cost is the same, array operations are often faster in practice. The linked list should be preferred where the asymptotic difference is truly large, for example for frequent insertion and deletion at a known position.

4.11 Common mistakes

Most errors in linked list code reduce to a few patterns. The list below collects them in one place; the details of each are explained in the relevant section.

1. **Overwriting a link without saving it.** In insertion in the middle, if `prev->link = fresh` is written first, the address of the old successor of `prev` is lost and the rest of the list is cut off. The rule: save first, then link.
2. **Passing the list by value to a routine that changes the head.** With a definition like `push_front(node_ptr head, …)` only the local copy changes; `list` in `main` is not updated and the new node leaks. If the head is going to change, the list is passed by address: `node_ptr *head` and `&list` in the call.
3. **Not checking the result of `malloc`; forgetting `free` or calling it twice.** An unchecked `malloc` writes to `NULL` when memory runs out. A forgotten `free` gives a leak, a second `free` gives undefined behavior.
4. **Reading the link field of a freed node.** `free(p); p = p->link;` reads from returned memory. The successor is saved first, then the node is freed.
5. **Not testing the boundary cases.** Operations on the empty list, a one-node list, the head node and the last node must be tried separately. Most bugs show up in one of these four cases.
6. **Traversing a circular list with `p != NULL`.** In a circular list no link is `NULL`; the loop never ends. The stopping condition is returning to the starting node, written with `do-while`.
7. **Writing the condition in the wrong order.** `p->data != x && p != NULL` looks inside `NULL` at the end of the list. To take advantage of short-circuit evaluation, the `NULL` check always comes first.

Most of these errors escape the compiler, and the program gives the right result on most inputs. List code should therefore be tested with small tests that cover the boundary cases and, where possible, with tools that catch memory errors. On Linux and macOS, the compiler option `-fsanitize=address` reports access after `free` and leaks at run time. What homework and exams ask for, beyond working code, is the ability to explain what every pointer points to at every moment.

4.12 Exercises and solutions

The five exercises in the last section of the slides apply the patterns learned in the chapter to new problems. Below, the solution idea and the full C code of each exercise are given. All of the code was compiled and run on sample lists. The node type and the `avail` functions are the same as in the chapter.

4.12.1 Merging two sorted lists without allocating nodes

Two sorted lists are to be merged into a single sorted list; no new node is to be allocated, only the links of the existing nodes rearranged. The idea is this: the nodes at the heads of the two lists are compared, the smaller one is linked to the end of the result list, and one step is taken in that list. When one of the lists runs out, the other is attached as it is; the remaining nodes are already sorted and all are greater than or equal to those in the result.

```
node_ptr sorted_merge(node_ptr a, node_ptr b) {
    struct node head;             /* temporary header node */
    node_ptr tail = &head;
    while (a != NULL && b != NULL) {
        if (a->data <= b->data) { tail->link = a; a = a->link; }
        else                    { tail->link = b; b = b->link; }
        tail = tail->link;
    }
    tail->link = (a != NULL) ? a : b;  /* attach the rest as is */
    return head.link;
}
```

The subtle part of the code is the local `struct node` variable `head`. It acts as a temporary header node. The first node of the result list is written into its link; the special case “is the result list still empty?” therefore disappears. The pointer `tail` holds the last node of the result list and advances by one after each linking. At the end the function returns the link of the temporary header, that is, the head of the real result list. Since the temporary header lives on the stack, it needs neither `malloc` nor `free`. Because `<=` is used in the comparison, for equal values the node of the first list comes first. The lists 1 → 4 → 7 and 2 → 3 → 8 → 9 become 1 → 2 → 3 → 4 → 7 → 8 → 9. Since one node is added to the result in each iteration, the time is *O(n + m)* and the extra memory is *O(1)*.

4.12.2 The *k*-th node from the end

The *k*-th node from the end of a list is to be found in a single pass (*k = 1* is the last node). Two pointers are used. The fast pointer is first moved *k* steps ahead. Then the two pointers are walked together, one step at a time. When the fast pointer reaches `NULL`, the slow pointer is exactly *k* nodes behind it, that is, on the *k*-th node from the end.

```
node_ptr kth_from_end(node_ptr head, int k) {
    node_ptr fast = head, slow = head;
    for (int i = 0; i < k; i++) {
        if (fast == NULL) return NULL;   /* list shorter than k */
        fast = fast->link;
    }
    while (fast != NULL) {
        fast = fast->link;
        slow = slow->link;
    }
    return slow;
}
```

If the list is shorter than *k*, the fast pointer falls onto `NULL` while being moved ahead and the function returns `NULL`. On the list 1 → 2 → 3 → 4 → 7 → 8 → 9, 9 was found for *k = 1*, 7 for *k = 3* and 1 for *k = 7*; for *k = 8*, `NULL` was returned. The fast pointer takes *n* steps in total and the slow pointer *n* − *k*; the time is $\Theta(n)$ and the extra memory is $O(1)$.

4.12.3 Removing duplicates from a sorted list

In a sorted list, equal values stand next to each other. Each node is therefore compared only with its successor. If they are equal, the successor is deleted and `p` stays where it is, because the new successor may carry the same value too. If they are not equal, `p` advances. The deleted nodes are returned to `avail` with `ret_node` instead of `free`.

```
void return_next(node_ptr prev) {
    node_ptr victim = prev->link;
    if (victim == NULL) return;
    prev->link = victim->link;
    ret_node(victim);
}
void remove_duplicates(node_ptr head) {
    node_ptr p = head;
    while (p != NULL && p->link != NULL) {
        if (p->data == p->link->data) return_next(p);  /* p stays put */
        else p = p->link;
    }
}
```

`return_next` is the form of the chapter's `delete_after` function that calls `ret_node` instead of `free`. The list 1 → 1 → 2 → 3 → 3 → 3 → 5 becomes 1 → 2 → 3 → 5, and the three deleted nodes go to `avail`. Each test either deletes a node or advances `p`; the total number of tests is $n - 1$ and the time is $O(n)$.

4.12.4 Recursive reversal

A recursive version of `reverse` rests on this thought: the list after the first node is reversed recursively, and then the first node is attached to the end of this reversed list. The end of the reversed list is the old second node, and the link of the first node still points to it.

```
node_ptr reverse_rec(node_ptr p) {
    node_ptr new_head;
    if (p == NULL || p->link == NULL) return p;  /* 0 or 1 node */
    new_head = reverse_rec(p->link);             /* reverse the rest */
    p->link->link = p;                           /* old next points back to me */
    p->link = NULL;
    return new_head;
}
```

The base case is the empty or one-node list; such a list equals its own reversal. Otherwise the rest of the list is reversed and the new head is kept in `new_head`. The line `p->link->link = p` turns the link of the old second node to `p`; `p->link = NULL` makes `p` the new last node. The list 10 → 20 → 30 → 40 becomes 40 → 30 → 20 → 10. The time is again $\Theta(n)$, but since a call frame is opened for each node, the extra memory is $\Theta(n)$. The $O(1)$ extra memory of the iterative version is therefore better. Because the recursive call is not the last thing the function does, the compiler cannot turn it into a loop either.

4.12.5 The length of the cycle

After a cycle has been found with Floyd's method, its length is to be computed. The meeting node is inside the cycle. A pointer starting there is walked, counting, until it returns to the same node.

```c
int cycle_length(node_ptr head) {
    node_ptr slow = head, fast = head;
    while (fast != NULL && fast->link != NULL) {
        slow = slow->link;
        fast = fast->link->link;
        if (slow == fast) {              /* meeting inside the cycle */
            int n = 1;
            for (node_ptr p = slow->link; p != slow; p = p->link) n++;
            return n;
        }
    }
    return 0;                                /* no cycle */
}
```

On the list 1 → 2 → 3 → 4 → 5, the function returned 3 when the link of 5 was redirected to 3, 5 when it was redirected to 1, and 1 when it was redirected to 5 itself; on a list without a cycle it returns 0. At most n iterations are taken up to the meeting, plus λ steps for counting; the time is $O(n)$ and the extra memory is $O(1)$.

4.13 Summary

- A **linked list** keeps its elements in nodes scattered in memory; the order is established by the **link** field of each node. The list is represented by a pointer to the first node; the **empty list** is `NULL`.
- A node is a **self-referential structure**: with `typedef struct node *node_ptr;` the pointer type is named first, and the structure carries a link field of its own type.
- Nodes are allocated from heap memory with `malloc(sizeof *p)`; the result is checked against `NULL`. For every `malloc`, exactly one `free` is called. A forgotten `free` gives a **memory leak**; an early or double `free` gives a **dangling pointer** and undefined behavior.
- **Link discipline**: save first, then link, free last. If the head of the list is going to change, the list is passed by address (`node_ptr *head`, `&list` in the call).
- In a singly linked list, insertion at the front is $O(1)$ and at the end $O(n)$; insertion after a known node and deleting the next node are $O(1)$, deletion by value and search are $O(n)$, and access to the i-th element is $O(i)$. Binary search is useless on a linked list.
- A list can be released recursively or with a loop; the recursive version uses $\Theta(n)$ stack and can cause stack overflow on a very long list.
- **Reversal** (three pointers), **concatenation**, the **middle node** (slow and fast pointers) and **Floyd's cycle detection** work in a single pass with $O(1)$ extra memory. In Floyd's method, if after the meeting one pointer is moved to the head and both are walked one step at a time, the start of the cycle is found.

- A **circular list** is held by its last node; insertion at the front and at the end is *O(1)*, and traversal is done with `do-while`. A **header node** removes the special case of the head node. A **doubly linked list** allows traversal in both directions and deletion of a known node in *O(1)*; its invariant is `p == p->llink->rlink == p->rlink->llink`.
- The **available-node list** (`avail`) reuses deleted nodes; `get_node` and `ret_node` are the pop and push operations of a stack. A circular list is returned to `avail` in *O(1)*.

4.14 Key concepts

The key terms of this chapter:

linked list singly linked list node link head pointer pointer to pointer null pointer (NULL) self-referential struct incomplete type typedef dynamic memory allocation heap memory malloc free memory leak dangling pointer short-circuit evaluation stack overflow tail recursion reversing a list cycle cycle detection circular linked list header node sentinel doubly linked list available space list push

---

4.15 Try It Yourself

1. The calls `push_back(&list, 3)`, `push_back(&list, 6)`, `push_back(&list, 9)`, `push_front(&list, 1)`, `push_back(&list, 12)` and `delete_value(&list, 6)` are applied in order to an empty list. What does `print_list(list)` print?

   Answer

   After the three insertions at the end the list is 3 → 6 → 9. Insertion at the front gives 1 → 3 → 6 → 9, and insertion at the end gives 1 → 3 → 6 → 9 → 12. `delete_value` finds 6 (its predecessor is 3) and turns the link of 3 to 9. The output is `1 3 9 12`.

2. Although `print_list(node_ptr p)` advances `p` until `NULL`, why does `list` in `main` not change? Why is `push_front` called with `&list`?

   Answer

   C passes arguments by value. `print_list` receives a copy of the address in `list`, and the function advances only this copy. `push_front`, on the other hand, must change the head of the list. Called by value, only the local copy would point to the new node, `list` in `main` would stay as it was and the new node would leak. With `&list` the function is given the address of the variable, and `*head = fresh` changes `list` directly.

3. A second 5 is inserted into the sorted list 2 → 5 → 9 with `sorted_insert`. When the loop ends, what is `prev`, and does the new 5 go in front of or behind the old one?

> **Answer**
>
> The first test 2 < 5 is true; `prev` moves to 2 and `p` to the old 5. The second test 5 < 5 is false and the loop ends. `prev` is node(2). The new 5 goes in front of the old 5: 2 → 5 (new) → 5 (old) → 9. Had the condition been `<=`, it would have gone behind.

4. Which node does the function `middle` return on the list 1 → 2 → 3 → 4 → 5 → 6? How many iterations does the loop run?

> **Answer**
>
> It runs three iterations. In them `slow` moves to 2, 3, 4; `fast` moves to 3, 5 and `NULL`. When `fast == NULL` the loop ends and the function returns 4, the second middle.

5. In the list 1 → 2 → 3 → 4 → 5 → 6, the link of 6 points to 4. In which iteration, and at which node, does `has_cycle` find the meeting? How many steps does the step that finds the start of the cycle take?

> **Answer**
>
> At the end of the iterations (`slow`, `fast`) are (2, 3), (3, 5), (4, 4) in turn; the meeting is in iteration 3, at 4. Here $\mu = 3$ and $\lambda = 3$. If one pointer is moved to the head and both are walked one step at a time, after 3 steps (1 → 2 → 3 → 4 and 4 → 5 → 6 → 4) both meet at 4. The start of the cycle is 4.

6. If, in the body of the loop in `reverse`, the lines `p = p->link` and `q->link = r` are swapped, what happens to the list 10 → 20 → 30?

> **Answer**
>
> In the first iteration `r = NULL` and `q = p` = node(10). `q->link = r` makes the link of 10 `NULL`. Since `q` and `p` point to the same node, the following `p = p->link` makes `p` `NULL`. The loop ends and `*list = q` leaves the list consisting of 10 alone. 20 and 30 are pointed to by no pointer; they have leaked.

7. If the last two lines of `dinsert` are swapped, that is, `p->rlink = fresh` is written first and then `p->rlink->llink = fresh`, what is the result of inserting 6 to the right of 4 in the list H ⇄ 4 ⇄ 8

⇄ H?

Answer

`p->rlink = fresh` makes the right of 4 equal to 6. Then, since `p->rlink` is now 6, `p->rlink->llink = fresh` turns the left of 6 toward 6 itself. The left of 8 is still 4. Forward traversal gives the order H, 4, 6, 8 correctly; backward traversal jumps from 8 to 4 and does not see 6. For 6 the equality `p->llink->rlink == p` is broken (the left of 6 is itself and its right is 8).

8. A program calls `get_node` three times with `avail` empty, then `ret_node` twice, then `get_node` three times again. How many times is `malloc` called in total?

Answer

In the first three calls `avail` is empty; three `malloc` calls are made. The two `ret_node` calls put two nodes into `avail`. Of the next three `get_node` calls, the first two take these nodes; at the third, `avail` is empty again and one more `malloc` is made. Four `malloc` calls in total.

9. In a singly linked list only the head pointer is kept. What is the cost of deleting the last node? Would keeping a `tail` pointer as well reduce this cost? What is the situation in a doubly linked list with a header?

Answer

Deleting the last node requires its predecessor, the second-to-last node, which can be found only by walking from the head: $\Theta(n)$. A `tail` pointer makes insertion at the end $O(1)$ but does not speed up deletion from the end; updating `tail` after the deletion also needs the second-to-last node. In a doubly linked list with a header, the last node is `header->llink` and its predecessor is `p->llink`; deletion is $O(1)$.

4.16 Further reading

- E. Horowitz, S. Sahni and S. Anderson-Freed, *Fundamentals of Data Structures in C* (2nd edition), Chapter 4: the source of the C implementations that these notes follow, of the `avail` list, of circular and doubly linked lists, and of polynomial operations with linked lists.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms*, Section 10.2: presents the doubly linked list and the sentinel node in pseudocode, concisely and precisely.
- D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, Sections 2.2.3–2.2.5: examines linked allocation, circular lists and doubly linked lists, together with the available-space list and their historical origins.

- D. E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, Section 3.1, Exercise 6: gives Floyd's cycle-finding method in the context of the period of random number generators.

4.17 References

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, MIT Press, Section 10.2 (lists with sentinels).
2. E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2nd edition, Chapter 4 (lists, `avail`, circular and doubly linked lists).
3. D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, 3rd edition, Addison-Wesley, 1997, Sections 2.2.3–2.2.5.
4. D. E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, Section 3.1, Exercise 6 (Floyd's cycle detection).

PART II · CHAPTER 5

Polynomials and Sparse Matrices

The value of a data structure depends on how it stores the same information in memory. These notes make that point with two familiar mathematical objects: the polynomial and the matrix. In both, most values may be zero. Of the one hundred and one coefficients of $5x^{100} + 1$, only two are nonzero. In the large matrices of scientific computing, the nonzero entries can be fewer than one in a thousand. A representation that stores every zero wastes both memory and time. The notes first store a polynomial as a coefficient array and as a term array, add two sorted term arrays by merging them, and multiply polynomials with linked lists. Then they represent a sparse matrix with triples, transpose it in two ways, and multiply two sparse matrices. The last part introduces compressed sparse row storage and the linked sparse matrix. Every program is given in C, traced step by step, and its cost is derived with a justification.

LEARNING OUTCOMES

Choosing a representation
Compare the coefficient array and the term array for polynomials in terms of memory and time.

Merge logic
Add two sorted term arrays with `padd` in $O(n + m)$ time and prove this bound.

Multiplication with linked lists
Multiply polynomials with a linked list: combine terms with equal exponents, delete zero terms, and return the memory.

Sparse matrices
Store a matrix as ⟨row, column, value⟩ triples and compute at what density this representation pays off.

Transposing
Trace `transpose` and `fast_transpose`; compare their costs of $O(cols \cdot terms)$ and $O(cols + terms)$.

Multiplying
Build and analyze sparse matrix multiplication with the help of the transpose; recognize compressed sparse row storage.

5.1 The polynomial abstract data type

The polynomial is familiar from high-school mathematics, and it turns up often in computing: error-detecting codes, cryptography, curve fitting and numerical integration all work with polynomials. To use this object in a program we first fix the operations we need, and then choose a memory layout that makes those operations fast. The first step is the definition of an **abstract data type** (ADT). An abstract data type describes the values of a data type and the operations on those values, but does not say how they are stored in memory. The memory layout is chosen in the second step, the implementation.

5.1.1 Polynomial, term and degree

Definition 5.1

A **polynomial** is an expression of the form $A(x) = a_n x^n + \ldots + a_1 x + a_0$. Each piece $a_i x^i$ is a **term**; a_i is the **coefficient** of the term and i is its **exponent**. The **degree** of the polynomial is the largest exponent with a nonzero coefficient. At the abstract level, a polynomial is a set of ⟨exponent, coefficient⟩ pairs in which each exponent occurs at most once.

The set view in the definition is important. The polynomial $A(x) = 3x^{20} + 2x^5 + 4$ is the set {⟨20, 3⟩, ⟨5, 2⟩, ⟨0, 4⟩}. Terms with zero coefficients do not enter the set: every term from x^{19} down to x^6 has coefficient zero, and none of them is written. The degree of the polynomial is 20. The constant term $4 = 4x^0$ has exponent 0. A polynomial with no terms at all is the **zero polynomial**; it corresponds to the empty set.

The rule that an exponent may not occur twice is also the basic condition for the addition and multiplication code. An expression like $2x^3 + 5x^3$ does not fit the set view; the two terms must be reduced to a single term $7x^3$. This reduction is what the multiplication code spends most of its effort on.

5.1.2 Operations

The textbook by Horowitz and coauthors defines the polynomial abstract data type with the operations in Table 5.1. The operation names are the names used in the book.

Operation	Meaning
`Zero()`, `IsZero(p)`	constructs the zero polynomial / tests whether p is the zero polynomial
`Coef(p, e)`	returns the coefficient of the term with exponent e; 0 if there is no such term
`LeadExp(p)`	returns the largest exponent, that is, the degree
`Attach(p, c, e)`, `Remove(p, e)`	adds the term $c{\cdot}x^e$ / removes the term with exponent e
`SingleMult(p, c, e)`	returns the product $p \cdot c{\cdot}x^e$

Operation	Meaning
`Add(p1, p2)`, `Mult(p1, p2)`	returns the sum / the product of two polynomials

Table 5.1: Operations of the polynomial abstract data type.

This list covers everything one does with polynomials. `Add` and `Mult` can be written with the other operations. Addition proceeds by comparing the largest exponents of the two polynomials with `LeadExp` and appending the appropriate term to the result with `Attach`. Multiplication calls `SingleMult` for each term of the first polynomial and adds the partial results. This is where the abstract data type pays off. The addition algorithm can be written without knowing whether the polynomial lives in memory as an array or as a linked list. A programmer who changes the implementation rewrites only these operations.

5.1.3 Rules for addition and multiplication

In addition, the coefficients of terms with equal exponents are added:

$$A(x) + B(x) = \Sigma\ (a_i + b_i) \cdot x^i$$

If an exponent occurs in only one polynomial, the coefficient of the other polynomial at that exponent counts as 0 and the term passes to the result unchanged. If the sum of two coefficients is 0, the term drops out of the result. For example, $(4x^3 + 2x^2 + 5) + (3x^3 - 2x^2 + x) = 7x^3 + x + 5$; the x^2 terms cancel.

In multiplication, every term is multiplied by every term of the other polynomial. When two terms are multiplied, **the exponents add and the coefficients multiply**: $a_i x^i \cdot b_j x^j = a_i b_j\, x^{i+j}$.

$$A(x) \cdot B(x) = \Sigma_i\ (\ a_i x^i \cdot \Sigma_j\ b_j x^j\ )$$

Multiplying a polynomial of n terms by a polynomial of m terms produces $n \cdot m$ products. Some of these products may have the same exponent. The product $(x + 1)(x - 1)$ produces four terms: x^2, $-x$, x and -1. The middle two terms have exponent 1; their sum is 0, so both vanish and the result is $x^2 - 1$. The real difficulty of the multiplication code is finding the terms with equal exponents, combining them, and discarding those that cancel.

5.2 Storing a polynomial in an array

The easiest way to store a polynomial in C is an array. But there are two different choices for what the array holds: reserve one cell for every exponent, or write down only the terms that exist. For the same polynomial, these two choices lead to very different memory and time costs.

5.2.1 First method: the coefficient array

The first method keeps all coefficients in an array indexed by exponent. `coef[i]` is the coefficient of x^i:

```
#define MAX_DEGREE 101   /* max degree + 1 */
typedef struct {
  int degree;               /* largest exponent */
  float coef[MAX_DEGREE];   /* coef[i]: coefficient of x^i */
} polynomial;
polynomial P;
```

The macro `MAX_DEGREE` is one more than the largest degree; an array of 101 cells holds the exponents 0 through 100. The `degree` field of the structure stores the largest exponent with a nonzero coefficient. Thanks to this field, the array need not be scanned from the end every time the degree is requested. The `typedef` gives the structure the name `polynomial`; the line `polynomial P;` declares a variable of this type.

The operations are very simple in this representation. `Coef(p, e)` is just `p.coef[e]` and takes *O(1)*. `LeadExp(p)` is `p.degree`. Addition adds the cells with the same index in the two arrays. Here is a short function that shows addition with coefficient arrays:

```
/* addition with coefficient arrays: d = a + b */
void array_add(const polynomial *a, const polynomial *b,
               polynomial *d) {
  int i, n = a->degree > b->degree ? a->degree : b->degree;
  for (i = 0; i <= n; i++) {
    float x = i <= a->degree ? a->coef[i] : 0;
    float y = i <= b->degree ? b->coef[i] : 0;
    d->coef[i] = x + y;
  }
  d->degree = n;
  while (d->degree > 0 && d->coef[d->degree] == 0)
    d->degree--;                /* drop leading zeros */
}
```

The function loops up to the larger of the two degrees. Once a polynomial's degree is exceeded, its coefficient is taken to be 0. The degree of the sum is first set to n; then the leading zeros are dropped. In the sum $(2x^3 + 1) + (-2x^3 + x + 4)$ the coefficient of x^3 comes out 0 and the last loop lowers the degree from 3 to 1; the result is $x + 5$. The running time is proportional to the larger of the two degrees: *O(max(deg A, deg B))*.

The problem is memory. Terms with coefficient 0 are stored too. The polynomial $P(x) = 5x^{100}$ has a single term, but the array reserves 101 cells and all of `coef[0..99]` are 0 (Figure 5.1). Since a `float` is 4 bytes, this structure occupies 408 bytes together with the `degree` field. The meaningful

information is only two numbers: the exponent 100 and the coefficient 5. Addition carries the same waste over into time: adding two one-term polynomials walks through 101 cells.



Figure 5.1: The coefficient array of the polynomial $5x^{100}$. Of the one hundred and one cells only the last is meaningful; the rest store zeros for nothing.

The coefficient array has one more limit. The degree cannot exceed `MAX_DEGREE - 1`. For a polynomial with a term x^{1000} the array must have one thousand and one cells; then every polynomial, even a one-term one, occupies four kilobytes. For a sparse polynomial, that is, one in which most coefficients are 0, this method is unsuitable. If most coefficients are nonzero, as in $x^4 + 3x^3 - x^2 + 2x + 7$, the coefficient array is both simple and fast.

5.2.2 Second method: the term array

The second method stores only the nonzero terms. Each term is a coefficient–exponent pair. The terms of all polynomials sit in a single shared array:

```
#define MAX_TERMS 101   /* max number of terms */
typedef struct {
  float coef;            /* coefficient */
  int expon;             /* exponent */
} polynomial;
polynomial P[MAX_TERMS]; /* all polynomials live here */
int avail = 0;           /* first free slot */
```

The structure is again named `polynomial`, but now it represents a single term: `coef` holds the coefficient and `expon` the exponent. The array `P` holds the terms of all polynomials in the program. The variable `avail` points to the first free slot in the array; a new term is always written to `P[avail]` and `avail` is incremented.

A polynomial is a contiguous range in `P`. Two indices are enough to locate it: the index of its first term, `start`, and the index of its last term, `finish`. The terms are kept sorted **from the largest exponent to the smallest**. This order is the key to the speed of the addition algorithm. Figure 5.2 shows where the polynomials $A(x) = 3x^{20} + 2x^5 + 4$ and $B(x) = x^5 + 10x^3 + 3x^2 + 1$ sit in the shared array.

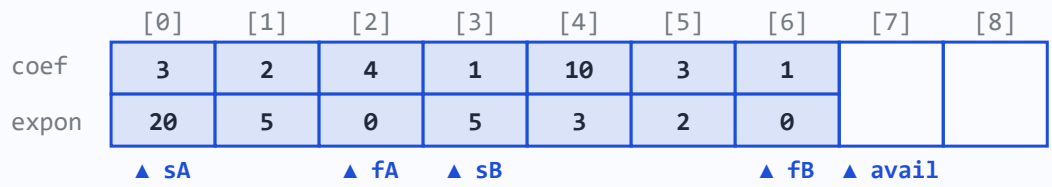


Figure 5.2: The shared array `P` holding the terms of two polynomials. *A* sits in `P[0..2]` (`sA = 0`, `fA = 2`), *B* in `P[3..6]` (`sB = 3`, `fB = 6`); the first free slot is `avail = 7`.

In this representation the polynomial $5x^{100}$ occupies a single cell: ⟨5, 100⟩. A term consists of a `float` coefficient and an `int` exponent and takes 8 bytes. Eight bytes suffice instead of the 408 bytes of the coefficient array. There is also no upper bound on the degree; the exponent can take any value that fits in an `int`. What is limited is the total number of terms of all polynomials: `MAX_TERMS` = 101.

An analogy explains the value of sorted terms. To turn two alphabetically sorted student lists into one sorted list, it is enough to walk down both lists at once. At each step, whichever of the two heads comes first is written to the new list. If the lists were unsorted, we would have to scan the whole other list for every name. This two-pointer walk is called **merging**. The course instructor stresses the same point: merge logic makes it possible to build more efficient algorithms on sorted structures. Polynomial addition rests entirely on this idea.

5.2.3 Comparing the two representations

Which representation is better depends on the structure of the polynomial. Table 5.2 compares the two in terms of memory and basic operations. In the table, d is the degree and t is the number of nonzero terms.

Criterion	Coefficient array	Term array
Memory	`MAX_DEGREE` cells (fixed)	t terms (8 bytes each)
`Coef(p, e)`	$O(1)$	$O(t)$ (or $O(\log t)$ with binary search)
`LeadExp(p)`	$O(1)$ (`degree` field)	$O(1)$ (first term)
Addition	$O(d)$	$O(t_A + t_B)$
Degree limit	`MAX_DEGREE - 1`	none (only the total term limit)
Suitable for	dense, low-degree polynomials	sparse polynomials

Table 5.2: Comparison of the coefficient array and the term array.

In the term array `Coef` becomes slower, because the requested exponent must be searched for in the array. But most polynomial operations walk through the terms in order instead of querying coefficients one by one. Addition and multiplication are such operations, and on the term array they take time proportional to the number of terms. Adding $3x^{20} + 2x^5 + 4$ with coefficient arrays walks through 21 cells; with the term array 3 terms are enough.

Common mistake

Writing a term with a zero coefficient into the term array, or adding the same exponent twice. All algorithms of this representation assume that every exponent occurs at most once and that the coefficients are nonzero. A term like $0x^5$ wastes space, reports the wrong degree and breaks the `IsZero` test: a polynomial whose only term is 0 is the zero polynomial, yet it does not look empty.

5.3 Polynomial addition: padd

The function that adds two polynomials in the term-array representation is called `padd` (polynomial add). It adds A in `P[sa..fa]` and B in `P[sb..fb]` and writes the result into the free part of `P`, starting at `avail`.

5.3.1 Merging with two pointers

The terms of both polynomials are sorted from the largest exponent to the smallest. The indices `sa` and `sb` point to the first unprocessed term of each polynomial; we call them **pointers**. They are array indices, not C pointers. At each step the exponents of the terms the two pointers point to are compared. There are three cases:

- **The exponents are equal.** The coefficients are added. If the sum is nonzero, a term with this exponent is appended to the result; if it is zero, the term vanishes. Both pointers advance by one.
- **A's exponent is larger.** B has no term with this exponent. All remaining terms of B have exponents no larger than B's current term, and the current term is smaller than A's. So this exponent occurs nowhere in B. A's term is appended to the result unchanged and only A's pointer advances.
- **B's exponent is larger.** The case is symmetric: B's term is appended and only B's pointer advances.

When the terms of one polynomial run out, the loop stops. The exponents of the remaining terms of the other polynomial are smaller than those of all terms already written to the result; these terms are copied to the result unchanged.

The justification of the second case is the heart of the correctness of the algorithm, and it holds only because of sortedness. If the terms were unsorted, we would have to scan all of B to find out whether a term of A has a counterpart in B. In a sorted array, B's current term alone answers the question.

5.3.2 COMPARE and attach

`padd` uses two helpers. The first is a macro that performs a three-way comparison, the second is a function that appends a term to the result:

```
#define COMPARE(x, y) \
  ((x) < (y) ? -1 : (y) < (x) ? 1 : 0)

void attach(float coef, int expon) {
  if (avail >= MAX_TERMS) {
    fprintf(stderr, "Array overflow!\n"); exit(1);
  }
  P[avail].coef = coef;
  P[avail++].expon = expon;
}
#define APPEND(k) attach(P[k].coef, P[k].expon)
```

`COMPARE(x, y)` yields *−1* if *x* < *y*, 1 if *x* > *y*, and 0 if they are equal. It is written with two nested conditional operators. The macro parameters are parenthesized at every use because the preprocessor substitutes the macro as text. In a call like `COMPARE(a + 1, b)`, the parentheses keep operator precedence from grouping the expression the wrong way. The three-valued result is what lets `padd` be written with a `switch`.

`attach` writes a term to `P[avail]` and increments `avail`. Before writing it checks whether the array is full; if so, it prints a message to the standard error stream and terminates the program. Without this check it would write to `P[101]`. Writing outside the bounds of an array is undefined behavior in C: neighboring variables may be corrupted or the program may crash. The line `P[avail++].expon = expon;` uses the postfix increment operator: first the exponent of `P[avail]` is written, then `avail` is incremented.

The macro `APPEND(k)` is shorthand for copying the term `P[k]` to the result unchanged. After preprocessing, `APPEND(sb)` becomes `attach(P[sb].coef, P[sb].expon)`.

Common mistake

Declaring the coefficient parameter of `attach` as `int`: `void attach(int coef, int expon)`. Then a `float` coefficient is truncated to an integer as it is passed to the function, and the coefficient 2.5 becomes 2. The compiler performs this conversion on its own; it gives no warning even with `-Wall -Wextra`. A warning appears only with the `-Wconversion` option. The type of the parameter must match the type of the field it is stored in: `float coef`.

5.3.3 The padd() code

```
void padd(int sa,int fa,int sb,int fb,int*sd,int*fd){
  float c; *sd = avail;              /* start of D */
  while (sa <= fa && sb <= fb)
    switch (COMPARE(P[sa].expon, P[sb].expon)) {
    case -1: APPEND(sb); sb++; break;     /* B larger */
    case  1: APPEND(sa); sa++; break;     /* A larger */
    case  0: c = P[sa].coef + P[sb].coef;
      if (c) attach(c, P[sa].expon); /* 0: skip */
      sa++; sb++;
    }
  for (; sa <= fa; sa++) APPEND(sa);   /* rest of A */
  for (; sb <= fb; sb++) APPEND(sb);   /* rest of B */
  *fd = avail - 1;
}
```

The function takes six parameters. The first four are the input ranges: A sits in `P[sa..fa]` and B in `P[sb..fb]`. The last two are outputs: `sd` and `fd` are the addresses of the variables that receive the start and end indices of the result. A C function can return only one value. To return two values, the caller passes the addresses of two variables and the function writes to those addresses. The expression `*sd = avail;` stores the current value of `avail` in the caller's variable.

The structure of the code is as follows:

1. **Line 2:** The first term of the result will be written to `P[avail]`, so `*sd` is fixed right away.
2. **Lines 3–10:** The loop runs as long as both polynomials have terms left. The `switch` picks one of three branches according to the comparison of the exponents. `case -1` appends the term of B when B's exponent is larger. `case 1` appends the term of A. `case 0` adds the coefficients; if the sum is not 0, `attach` is called. The condition `if (c)` means `c != 0`.
3. **Lines 11–12:** When the loop ends, at most one of the two polynomials has terms left. Only one of the two `for` loops does any work; the other finds its condition false from the start.
4. **Line 13:** The last term appended is in `P[avail - 1]`; `*fd` receives this index.

The `case 0` branch has no `break` at its end. Since this is the last branch of the `switch`, control falls out of the `switch` anyway; it does not run into another branch. Still, writing a `break` in the last branch too is a good habit: if a new branch is added below it later, the code will not flow into it by accident.

5.3.4 Code trace: A + B

The polynomial $A(x) = 4x^3 + 2x^2 + 5$ sits in `P[0..2]`, $B(x) = 3x^3 - 2x^2 + x$ sits in `P[3..5]`, and `avail = 6` (Figure 5.3). The call `padd(0, 2, 3, 5, &sd, &fd)` proceeds as in Table 5.3.

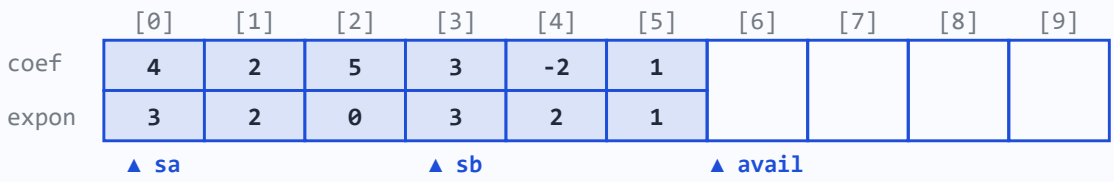


Figure 5.3: The start of `padd`. `sa = 0` points to the first term of *A*, `sb = 3` to the first term of *B*; the result will start at `avail = 6`.

Step	sa	sb	Exponents	COMPARE	Action	avail
0	0	3	—	—	`*sd = 6`	6
1	0	3	3, 3	0	$c = 4 + 3 = 7$; `attach(7, 3)`: `P[6] = 7x³`; both advance	7
2	1	4	2, 2	0	$c = 2 + (-2) = 0$; not appended; both advance	7
3	2	5	0, 1	−1	`APPEND(5)`: `P[7] = 1x¹`; only `sb` advances	8
4	2	6	—	—	`sb = 6 > fb = 5`: the loop ends	8
5	3	6	—	—	rest of A: `APPEND(2)`: `P[8] = 5x⁰`	9
6	3	6	—	—	`*fd = 9 - 1 = 8`	9

Table 5.3: Trace of `padd(0, 2, 3, 5, &sd, &fd)`.

In the table, the `sa` and `sb` columns show the values at the start of the step, and the `avail` column shows the value at the end of the step. In the first two rounds the exponents are equal. In the first, the coefficients are added and $7x^3$ is written. In the second, $2x^2$ and $-2x^2$ cancel; since `c` is zero, `attach` is not called and `avail` does not change. In the third round the remaining term of *A* is $5x^0$ and that of *B* is x^1. Because `COMPARE(0, 1) = -1`, the term of *B* is appended. After this step *B* is exhausted and the loop condition becomes false. The remaining *5* of *A* is copied by the first `for` loop.

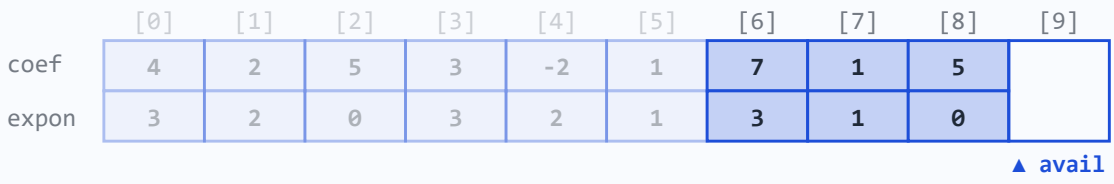


Figure 5.4: When `padd` finishes: the result $D(x) = 7x^3 + x + 5$ sits in `P[6..8]` and `avail = 9`. The faded cells are the input polynomials.

The result is $D(x) = 7x^3 + x + 5$ and it occupies `P[6..8]`: `sd = 6`, `fd = 8`. When the program is run on this input, `attach` is called three times and `avail` becomes 9. The input polynomials stay where they are; `padd` does not modify them.

Example 5.1

Adding the polynomials of Figure 5.2: $A = 3x^{20} + 2x^5 + 4$ (`P[0..2]`), $B = x^5 + 10x^3 + 3x^2 + 1$ (`P[3..6]`), `avail = 7` .

1. 20 versus 5: A is larger; $3x^{20} \rightarrow$ `P[7]` .
2. 5 versus 5: equal; $2 + 1 = 3$, $3x^5 \rightarrow$ `P[8]` .
3. 0 versus 3: B is larger; $10x^3 \rightarrow$ `P[9]` .
4. 0 versus 2: B is larger; $3x^2 \rightarrow$ `P[10]` .
5. 0 versus 0: equal; $4 + 1 = 5 \rightarrow$ `P[11]` . Both polynomials are exhausted; the remaining copy loops do nothing.

The result is $3x^{20} + 3x^5 + 10x^3 + 3x^2 + 5$ and it sits in `P[7..11]` . The loop runs five rounds and makes five `attach` calls. With coefficient arrays the same addition would walk through 21 cells.

5.3.5 Complexity

Theorem 5.1 (Cost of padd)

If A has n terms and B has m terms, `padd` runs in $\Theta(n + m)$ time and writes at most $n + m$ terms.

Proof. *Upper bound.* In every round of the `while` loop, at least one of `sa` and `sb` increases by one. `sa` can increase at most n times and `sb` at most m times. So the loop runs at most $n + m$ rounds. The two `for` loops that copy the remaining terms after the loop increment the same pointers. The total increase of the pointers cannot exceed $n + m$. So the total number of rounds of all loops is also at most $n + m$. Each round does a constant amount of work (one comparison, at most one addition and one `attach`). The time is $O(n + m)$.

Lower bound. Each round advances at most two pointers by one each. For all terms to be processed, the pointers must advance $n + m$ times in total; that means at least $(n + m)/2$ rounds. The time is $\Omega(n + m)$. Together the two bounds give $\Theta(n + m)$. □

The best case is when all exponents are shared ($n = m$). Every round enters `case 0` and both pointers advance together; the loop runs n rounds. The cost is then $\Omega(n)$; since $n = m$, this is of the same order as $\Theta(n + m)$. The worst case is when there is no common exponent. Then every term is written separately and $n + m$ calls to `attach` are made. That is why adding two ten-term polynomials makes at most 20 calls to `attach` . This count has also been verified by a program. Adding a ten-term polynomial with odd exponents to a ten-term polynomial with even exponents counts 20 calls. Adding two identical polynomials counts 10 calls.

What would happen if the terms were kept unsorted? For each term of A we would have to search B for a term with the same exponent, and if none were found, put the term in the right place in the result. The search takes $O(m)$ per term and $O(nm)$ for n terms. If insertions are used to keep the result sorted, every insertion shifts the terms in the array, and the total rises to the order of $O((n + m)^2)$. As the instructor puts it, if $O(n + m)$ is what we want and we can do better, so much the better. But since every term of the input must be looked at least once, no algorithm for addition can be faster than $n + m$; `padd` is of optimal order.

5.3.6 The price of the shared array

The shared array `P` is a simple memory manager: `avail` only moves forward. Every call to `padd` appends its result to the end of the array. The terms of a polynomial that was only needed as an intermediate result stay in the array, and their space is not reclaimed. In a long computation the array fills up with such leftovers and `attach` stops the program with the message “Array overflow!”, even though the live polynomials in the array may be small in total.

There are two solutions to this problem. The first is **compaction**. When the array is full, the polynomials still in use are shifted toward the start of the array and their start and end indices are updated. Compaction takes time proportional to the number of live terms and must be written carefully because it changes the indices of all polynomials. The second solution is to store the polynomial as a linked list. Each term sits in its own node; a node that is no longer needed is returned at once with `free`.

5.4 Multiplication on arrays and operations on many polynomials

`padd` adds two polynomials. Multiplication, and the sum and product of many polynomials, can also be written with the same array representation. The solutions to these three problems, which the slides leave as homework, are given in detail below.

5.4.1 pmult(): multiply by one term and add

The multiplication rule suggests an algorithm. For each term $a_i x^{e_i}$ of A, multiply B by this single term (`SingleMult`). The resulting polynomial of m terms is added to the partial result with `padd`. Since the terms of B are sorted, the product with a single term is also a sorted polynomial: the same e_i is added to every exponent and the order is preserved. The function written in these notes as the solution of the homework is the following:

```c
/* A = P[sa..fa], B = P[sb..fb]; product P[*sd..*fd] */
void pmult(int sa, int fa, int sb, int fb,
           int *sd, int *fd) {
  int i, j, ss = avail, sf = avail - 1;  /* empty partial */
  int ts, tf, ys, yf;
  for (i = sa; i <= fa; i++) {
    ts = avail;                      /* B · a_i x^e_i */
    for (j = sb; j <= fb; j++)
      attach(P[i].coef * P[j].coef,
             P[i].expon + P[j].expon);
    tf = avail - 1;
    padd(ss, sf, ts, tf, &ys, &yf);  /* add to partial */
    ss = ys;  sf = yf;
  }
  *sd = ss;  *fd = sf;
}
```

The partial result is `P[ss..sf]`. It starts out empty: since `sf = ss - 1`, the range contains no terms. In each round of the outer loop, $B \cdot a_i x^{e_i}$ is written between `ts` and `tf`. Then `padd` adds the old partial result to this temporary polynomial and puts the new partial result at the end of the array. In the first round the partial result is empty, so the main loop of `padd` never runs and the temporary polynomial is simply copied.

Example 5.2

$A = x + 1$ in `P[0..1]`, $B = x - 1$ in `P[2..3]`; `avail = 4`, the partial result is empty (`ss = 4`, `sf = 3`).

1. $i = 0$ (term x): $x \cdot B = x^2 - x \rightarrow$ `P[4..5]`. `padd(4, 3, 4, 5)` adds it to the empty partial result; the new partial result is `P[6..7]` $= x^2 - x$.
2. $i = 1$ (term 1): $1 \cdot B = x - 1 \rightarrow$ `P[8..9]`. `padd(6, 7, 8, 9)`: x^2 is appended, $-x + x = 0$ is skipped, -1 is copied. The partial result is `P[10..11]` $= x^2 - 1$.

The function returns `sd = 10`, `fd = 11` and `avail` becomes 12. Of the twelve positions only two hold the result; `P[4..9]` are now unused intermediate polynomials.

For the time analysis, let A have n terms and B have m terms. In round i the temporary polynomial has m terms and is written in $O(m)$ time. The partial result has at most $i \cdot m$ terms, since $i \cdot m$ products have been produced so far. `padd` takes $O(im + m)$ time in this round. Summed over all rounds:

$$\Sigma_{i=1}^{n} O(im + m) = O(m \cdot n(n + 1)/2 + nm) = O(n^2 m)$$

The form $O(n \cdot (nm + m)) = O(n^2 m)$ in the slides reaches the same result by bounding the partial result in every round by its largest size, nm. The result is not symmetric: n appears squared and m to the first power. If the smaller polynomial is put in the outer loop, the cost is $O(\min(n, m)^2 \cdot \max(n, m))$. Suppose a polynomial of 2 terms is multiplied by a polynomial of 100 terms. Putting the two-term one outside does about $2^2 \cdot 100 = 400$ units of work; the reverse does $100^2 \cdot 2 = 20{,}000$ units.

The memory cost is even worse. Each round writes m temporary terms and at most im new partial-result terms, and none of them is returned. The total consumption is also $O(n^2 m)$. This price of the shared array makes compaction or a linked list necessary in long computations. In the program, the product $(3x^{20} + 2x^5 + 4)(x^5 + 10x^3 + 3x^2 + 1)$ gives the correct 11-term result, but `avail` rises from 7 to 42: of the 35 positions only 11 are the result.

5.4.2 Adding n polynomials

A program that works with many polynomials can keep the location of each polynomial in two tables. `start[k]` is the index of the first term of polynomial k in `P`, and `finish[k]` is the index of its last term. The easiest way to add n polynomials is to call `padd` repeatedly: first the first two are added, then the third is added to the result, and so on.

Let N be the total number of terms in all the polynomials. In repeated addition the partial result grows and every `padd` walks over it from the beginning. In the worst case the terms of the first polynomial are copied $n - 1$ times. The total cost is $O(n \cdot N)$. For example, 100 polynomials of 100 terms each, with no shared exponents, require about $100 \cdot 10{,}000 / 2 = 500{,}000$ term copies.

A better way is to merge all n polynomials at once. One pointer is kept for each polynomial. At every step, the term with the largest exponent among the terms the n pointers point to is selected; terms with equal exponents are added. Finding the largest among n terms by a plain scan takes $O(n)$ per step and $O(n \cdot N)$ in total; there is no gain. Selecting the largest exponent with a **priority queue**, for example a heap, brings each step down to $O(\log n)$. The total cost is $O(N \log n)$. The same result is obtained by adding the polynomials two at a time, like a tournament tree. In each round the number of polynomials halves; each round does $O(N)$ work in total and there are $\lceil \log_2 n \rceil$ rounds.

5.4.3 Multiplying n polynomials

In multiplication the partial results grow quickly. If n polynomials are multiplied from left to right, the product P_1P_2 is multiplied by P_3, that result by P_4, and so on. At every step a large partial result is multiplied by a small polynomial. Since the cost of `pmult` grows with the square of the number of terms of the polynomial in the outer loop, the large partial result should always be put in the inner loop.

Multiplying two at a time, in a tree shape, balances the sizes. First P_1P_2, P_3P_4, … are multiplied; then these products are multiplied two at a time. At each level the polynomials being multiplied are of similar size. The same `start` / `finish` tables can be used to hold the partial results; but since leftovers pile up quickly in the array representation, a linked list is a more suitable choice.

Common mistake

Writing the product term by term directly at the end of the array without combining terms with equal exponents. If $x^2 - x + x - 1$ is written for $(x + 1)(x - 1)$, the array is no longer sorted and the same exponent occurs twice. Every later call to `padd` gives a wrong result with this polynomial, because the merge logic assumes that every exponent occurs once and in sorted order. That is why `pmult` adds every single-term product with `padd`.

5.5 Polynomials as linked lists

The array representation has two weak points: the total number of terms must be known in advance, and the space of terms that are no longer needed is not reclaimed. A linked list solves both. Each term sits in a node allocated with `malloc`. The nodes are tied together with pointers, and the number of terms can grow and shrink freely while the program runs. When a term is no longer needed, its node is returned to the system with `free`.

5.5.1 The node structure

```
typedef struct poly_node *poly_ptr;
typedef struct poly_node {
  int coef;
  int expon;
  poly_ptr link;
} poly_node;
```

The first line defines a pointer type to `struct poly_node` under the name `poly_ptr`. This line comes before the definition of the structure. C allows this, because declaring a pointer to a structure does not require knowing its contents; it is enough to know that the name denotes a structure. The definition that follows gives the three fields of the node: `coef`, `expon`, and `link`, which holds the address of the next node in the list. Because the structure contains a field that points to itself, it is called a **self-referential structure**. In this representation the coefficients are integers.

The list again keeps the terms in decreasing order of exponent and contains only nonzero terms. The address of the first node of the list is stored in a `poly_ptr` variable; the `link` field of the last node is `NULL`. Figure 5.5 shows the list of the polynomial $a = 3x^{14} + 2x^8 + 1$.

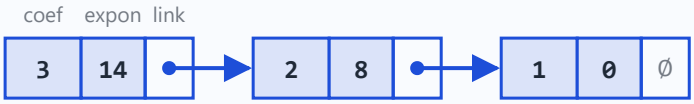


Figure 5.5: The linked list of the polynomial $a = 3x^{14} + 2x^{8} + 1$. Each node holds a coefficient, an exponent and the address of the next node; the link of the last node is `NULL` (∅).

On a 64-bit system a node occupies 16 bytes: 8 bytes for the two `int` fields and 8 bytes for the pointer. That is twice the space of the 8-byte term in the term array; this is the price of the link. In return, the nodes need not be contiguous in memory, and there is no limit of a shared array. The number of terms of a polynomial is limited only when the system's memory runs out.

The list has two important advantages. First, insertion and deletion in the middle are done by changing just a few pointers, in *O(1)* time; once the position has been found, of course. In an array, insertion in the middle would require shifting all following terms by one position. Second, addition is again a merge: one walks down the heads of both lists at once and applies the three cases of `padd` unchanged. Addition with linked lists also runs in *O(n + m)* time. The only difference is that the result nodes are allocated with `malloc` and returned with `free` when the work is done.

5.5.2 add_term() and delete_next()

The multiplication code relies on two helpers. The first creates a new term node and, optionally, links it after a given node:

```
/* create a term; if prev is given, link it after prev */
poly_ptr add_term(int expon, int coef,
                  poly_ptr prev) {
  poly_ptr p = malloc(sizeof *p);
  if (!p) { printf("Out of memory!\n"); exit(1); }
  p->expon = expon;  p->coef = coef;
  if (prev) {              /* in the middle or at the end */
    p->link = prev->link;    /* new node first */
    prev->link = p;         /* then prev */
  } else p->link = NULL;      /* standalone */
  return p;
}
```

`malloc(sizeof *p)` allocates as much memory as the type `p` points to, that is, `poly_node`. Writing `sizeof(poly_node)` is also correct; but the form `sizeof *p` stays correct on its own if the type of `p` changes. If `malloc` cannot find memory, it returns `NULL`. The code checks this case at once and

terminates the program with a message. Without the check, the next line would try to write to the place the `NULL` pointer points to; that is undefined behavior and on most systems the program crashes.

After the fields of the node are filled in, two cases are distinguished. If `prev` is given, the new node is linked after it. The order matters: first the link of the new node is pointed at the old successor of `prev`, then the link of `prev` at the new node. If the two lines were swapped, `prev->link` would be overwritten with the new node first; the address of the old successor would be lost and the rest of the list would become unreachable. If `prev` is `NULL`, the new node becomes a list on its own; its link is `NULL`. In every case the function returns the address of the new node.

The second helper deletes the node after a given node:

```
/* delete the node after prev */
void delete_next(
        poly_ptr prev) {
  poly_ptr victim =
        prev->link;
  if (!victim) return;
  prev->link =
        victim->link;
  free(victim);
}
```

The address of the node to delete is first saved in the variable `victim`. Then the link of `prev` is pointed at the successor of the node being deleted; this takes the node out of the list. Finally the memory of the node is returned with `free`. If the link were updated without saving the address, the node could never be reached again and its memory could never be returned. Memory lost this way is called a **memory leak**. As the instructor puts it, the leftovers in memory must be cleaned up. The reverse order is also wrong: if `free(prev->link)` is called first and then `prev->link = prev->link->link` is written, the fields of a freed node are read. A pointer that points to returned memory is called a **dangling pointer**, and using it is undefined behavior.

If there is no node to delete (`prev` is the last node), the function returns without doing anything. Note that the function cannot delete the first node: deletion is always done through the previous node. The multiplication code can live with this restriction, because the first term of the product list is never deleted.

5.5.3 pmult_list() and place()

In the product of two polynomials, every term of A is multiplied by every term of B. Each product produced is put in its place in the sorted result list. If a term with the same exponent already exists, its coefficient is updated; if the updated coefficient becomes 0, the term is deleted. For readability, this body of the inner loop is moved into a separate function:

```
poly_ptr place(poly_ptr r,
               int expon, int coef) {
  while (r->link && r->link->expon > expon) r = r->link;
  if (!r->link || r->link->expon < expon)
    add_term(expon, coef, r);          /* new exponent */
  else if ((r->link->coef += coef) == 0)
    delete_next(r);                    /* became 0 */
  return r;
}
```

`place` starts at node `r` and looks for the position of the new term. The assumption is that the exponent of `r` is larger than the exponent of the new term, that is, the new term will come somewhere after `r`. The `while` loop advances as long as the exponent of the next node is larger than the new exponent. When the loop ends, `r` is the last node that will come before the new term. Three cases remain:

- If there is no node after `r` (`!r->link`), or the exponent of the next node is smaller than the new exponent, this exponent is not in the list. `add_term(expon, coef, r)` links the new node after `r`.
- If the exponent of the next node equals the new exponent, `coef` is added to its coefficient. The expression `(r->link->coef += coef) == 0` first performs the addition and then compares the result with 0.
- If the sum became 0, `delete_next(r)` deletes that node.

The function returns `r`. This return value is the starting point of the next search. Since the terms of B arrive in decreasing order, the products produced with the same p are also in decreasing order. The next product comes after the position of the previous one in the list; there is no need to restart the search from the head of the list.

The multiplication function itself is the following:

```
poly_ptr pmult_list(poly_ptr p_head,
                    poly_ptr q_head) {
  poly_ptr p, q, r = NULL, anchor = NULL, r_head = NULL;
  for (p = p_head; p; p = p->link) {
    r = anchor;            /* start the search here */
    for (q = q_head; q; q = q->link) {
      int expon = p->expon + q->expon;
      int coef = p->coef * q->coef;
      if (r_head) r = place(r, expon, coef);
      else r_head=r=anchor=add_term(expon, coef, NULL);
      if (q == q_head) anchor = r;  /* for the next p */
    }
  }
  return r_head;
}
```

`p_head` and `q_head` are the heads of the lists of the two polynomials. `r_head` holds the head of the result list; the function returns it at the end. `r` is the position that walks through the result list in the inner loop. `anchor` is the node where the search for the next *p* will start.

The variable `anchor` rests on an observation. The product of *p* with the first term of *B*, `q_head`, is the term with the largest exponent that *p* can produce. The next term *p′* has a smaller exponent than *p*. All products of *p′* have exponents smaller than the product of *p* with `q_head` and cannot fall in front of it in the list. So the search for *p′* can start from where the product $p \cdot q_head$ was placed. In the code, this is done by saving in `anchor` the `r` that is returned when `q == q_head`. Each round of the outer loop starts with `r = anchor`.

While the result list is empty (`r_head == NULL`), `place` cannot be called, because `r` is not a valid node. In this case the first product becomes a standalone node with `add_term(expon, coef, NULL)`, and `r_head`, `r` and `anchor` are all pointed at this node. In the chained assignment `r_head=r=anchor=…` the assignment operator works from right to left: first `anchor`, then `r`, and last `r_head` receives the value.

The first node of the list is the product of the first term of *A* and the first term of *B*. The exponent of this term is the largest of all products and it is obtained only once: no other pair of terms can give the same exponent. So the first node is never updated or deleted; the fact that `delete_next` cannot delete the first node causes no problem. For the same reason, the exponent of the node `r` at which a search starts is always larger than the exponent of the new term, and the assumption of `place` is preserved.

5.5.4 Code trace: (x + 1)(x − 1)

The lists $A = x + 1$ and $B = x - 1$ have two nodes each. Nodes are written as (coefficient, exponent) pairs: $A = (1, 1) \to (1, 0)$, $B = (1, 1) \to (-1, 0)$. The expected result is $x^2 - 1$. Table 5.4 traces each of the four products. In the table, `&(1, 2)` is the address of the node with coefficient 1 and exponent 2.

Step	*p*, *q*	`expon`, `coef`	Action	Result list	`r`, `anchor`
1	*x*, *x*	2, 1	list empty: `add_term(2, 1, NULL)`; `q == q_head` → `anchor = r`	x^2	&(1, 2), & (1, 2)
2	*x*, *−1*	1, −1	`place`: no node after `r` → new node *−x*	$x^2 - x$	&(1, 2), & (1, 2)
3	*1*, *x*	1, 1	`r = anchor`; the next node has exponent 1 = 1 → coefficient *−1 + 1 = 0* → `delete_next(r)`; `anchor = r`	x^2	&(1, 2), & (1, 2)
4	*1*, *−1*	0, −1	`place`: no node after `r` → new node *−1*	$x^2 - 1$	&(1, 2), & (1, 2)

Table 5.4: Trace of `pmult_list(A, B)`, $A = x + 1$, $B = x - 1$.

In the first step the list is empty and the first node is created directly. In the second step `place` starts from `r = &(1, 2)`. Since there is no next node, the loop never runs and the node *−x* is attached after x^2 (Figure 5.6a). The function returns not the new node but its predecessor; `r` is still `&(1, 2)`.

The third step is the interesting moment of the multiplication. The outer loop moves on to $p = 1$ and starts with `r = anchor`. In this example `anchor` is the first node of the list; for longer polynomials the search can start from the middle of the list. The new product is $1 \cdot x = x$. The exponent of the next node, $-x$, is also 1. The coefficients are added: $-1 + 1 = 0$. Since the term has become zero, `delete_next(r)` removes it from the list and returns its memory (Figure 5.6b). In the fourth step the constant term $1 \cdot (-1) = -1$ is appended at the end of the list.

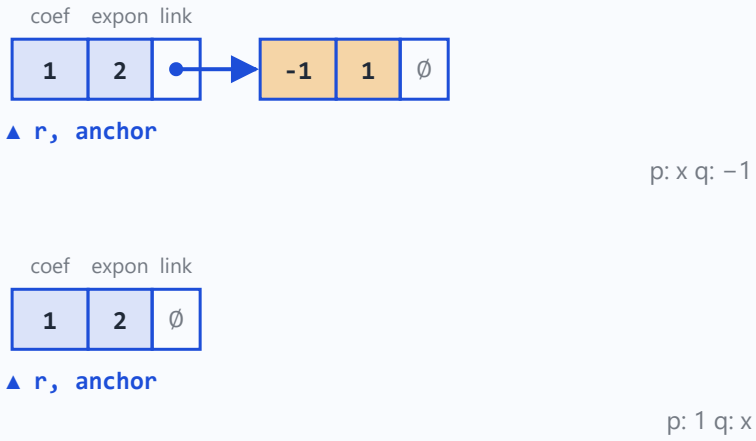


Figure 5.6: Two moments of `pmult_list`. (a) Step 2: the node $-x$ has been attached after x^2. (b) Step 3: when $+x$ is added, the coefficient becomes 0 and the node is deleted; the list is again a single node.

The result is $x^2 - 1$. $n \cdot m = 4$ products were made, and $-x$ and $+x$, having the same exponent, were combined and deleted. The function returns `r_head`. When the program is done with the result, it must return all nodes of the list one by one with `free`; the same holds for the inputs. The test driver has a short loop that does this: the address of the next node is saved, the current node is freed, and the saved address is moved to.

Example 5.3

Which term is deleted in the product $(x^2 + 1)(x^2 - 1)$?

1. $p = x^2$: $x^2 \cdot x^2 = x^4$ becomes the first node. $x^2 \cdot (-1) = -x^2$ is attached after it. The list: $x^4 - x^2$.
2. $p = 1$: the search starts from the node `anchor` $= x^4$. $1 \cdot x^2 = +x^2$; a node with the same exponent exists and the coefficient becomes $-1 + 1 = 0$. The x^2 term is deleted.
3. $1 \cdot (-1) = -1$ is appended at the end.

The result is $x^4 - 1$. The program produces the same list.

5.5.5 Complexity and measurement

Let A have n terms and B have m terms. The number of products is exactly nm. The work of each product itself is constant (one addition, one multiplication, at most one `malloc` or `free`). The real cost is the walk in the `while` loop of `place`.

For one term p, the inner loop starts at `anchor` and only moves forward in the list. `place` continues from where it left off on every call and never goes back. So the total walk for one p cannot exceed the length of the list after `anchor`. The result list has at most nm nodes. The cost of one p is $O(m + nm)$, and the cost of all p is $O(n^2m)$. This bound is the same as that of `pmult`, but the linked list spends no extra memory on partial results: the result list has at most nm nodes and nodes that become zero are returned at once.

Whether the bound is actually reached depends on the exponents of the terms. To see this, a counter was added to the loop of `place` and two polynomials with $n = m$ terms were multiplied in two different arrangements. In both cases the result consists of nm terms with distinct exponents:

- **Sparse A, dense B.** The exponents of A are $(n - 1)m, \ldots, m, 0$; those of B are $m - 1, \ldots, 0$. The products of each p form a block, and the block of the next p comes entirely after it.
- **Dense A, sparse B.** The exponents of A are $n - 1, \ldots, 0$; those of B are $(m - 1)n, \ldots, n, 0$. The products of each p are interleaved among the earlier ones.

$n = m$	Sparse A, dense B	Dense A, sparse B	$n^2m / 2$
10	178	502	500
20	758	4,007	4,000
40	3,118	32,017	32,000
80	12,638	256,037	256,000
160	50,878	2,048,077	2,048,000

Table 5.5: Total number of steps of the `place` loop ($n = m$, nm terms in the result).

In the first arrangement the number of steps quadruples when n doubles; it is about $2nm$, that is, $O(nm)$. Here `anchor` does its job: the search for each p starts at the head of the previous block and the list is traversed only once. In the second arrangement the number of steps grows eightfold and is almost equal to $n^2m/2$. Because the terms each p adds are scattered among the earlier terms, each p walks over most of the list again. This shows that the bound $O(n^2m)$ is actually reachable. The worst-case cost is $\Theta(n^2m)$.

The practical lesson from the measurement is this: if n and m differ, the smaller polynomial should go in the outer loop. Since the square of the number of terms of the outer polynomial enters the cost, putting a 2-term polynomial outside is about fifty times cheaper than putting a 100-term one outside.

The correctness of the code was tested as well. The `pmult_list` results for 20,000 randomly generated pairs of polynomials were compared with a brute-force product that accumulates the coefficients in an array. Every result list was found to have decreasing exponents, no zero coefficients, and to equal the brute-force result.

Common mistake

Leaving a term whose coefficient has dropped to 0 in the list. If the product $(x + 1)(x - 1)$ stays as the list $x^2 + 0x - 1$, the list carries an extra node, the degree and the number of terms look wrong, and later operations visit this node needlessly. A node that becomes zero must both be removed from the list and returned with `free`; changing only the link leads to a memory leak.

5.5.6 Other solutions to the problem

`pmult_list` is not the only way to multiply two polynomials. Table 5.6 gives the methods listed in the slides together with their complexities. A has n terms and B has m terms.

	Method	Complexity
1	Multiply everything and append each product at the end; then sort and merge	$O(n^2m^2)$
2	Multiply everything; for each product scan the result from the start and insert at the right position	$O(n^2m^2)$
3	Build a list for the product with each term of A and merge it with the result at every step	$O(n^2m)$
4	Method 3, smaller polynomial outside	$O(\min(n, m)^2 \cdot \max(n, m))$
5	A separate list for each term of A; at the end all are merged two at a time	$O(m \cdot n \cdot \lg n)$
6	Method 5, smaller polynomial outside	$O(m \cdot n \cdot \lg \min(m, n))$
7	Allocate an array; index k holds the coefficient of exponent k	—

Table 5.6: Different solutions to polynomial multiplication.

The first method writes the nm products into an unsorted list, then sorts the list and combines neighboring equal exponents. If the sort is done with a simple quadratic algorithm, for example insertion sort, $(nm)^2 = n^2m^2$ comparisons are needed. The second method scans the result from the start for every product; since the result can grow to nm terms, this is again $O(n^2m^2)$. The third method is the list version of `pmult`; `pmult_list` also comes down to this bound thanks to `anchor`.

The fifth method resembles the merge phase of merge sort. n lists are produced, each with m terms. Then they are merged two at a time: n lists drop to $n/2$, $n/2$ lists drop to $n/4$. At each level at most nm terms are processed in total and there are $\lceil \lg n \rceil$ levels. The total cost is $O(nm \lg n)$. Building the lists from the terms of the smaller polynomial puts the small number inside the logarithm.

The seventh method accumulates the result in a coefficient array: the coefficient of each product is added to `d[expon]`. The multiplication itself takes *O(nm)* and is extremely simple. But the size of the array depends on the size of the exponents. For two sparse polynomials containing $x^{1000000}$, an array of two million cells is needed. Converting the result array back into a term list also requires scanning the whole array. The course instructor gives a short and blunt answer to this idea: never an array whose *n*-th index holds the *n*-th exponent. Putting a non-sparse array inside a structure designed for sparse data gives back all the gains of the structure. The test program of these notes used the seventh method only as a correctness check, with small exponents.

5.6 Sparse matrices

The problem seen with polynomials appears on a larger scale in two-dimensional structures. The adjacency matrix of a city's road network, the friendship relations in a social network, or the system of equations in an engineering simulation are matrices with thousands of rows and columns. The vast majority of their entries are zero: each junction is connected to a few junctions, and each person's number of friends is small next to the total population.

5.6.1 Definition

> **Definition 5.2**
>
> A **sparse matrix** is a matrix most of whose entries are zero. The nonzero entries are called **nonzero entries**. There is no exact threshold for calling a matrix sparse. In practice a matrix is considered sparse if the number of nonzero entries t is a small fraction of the total number of entries, $rows \cdot cols$.

Of the 36 cells of the 6×6 matrix in Figure 5.7, only 8 are nonzero. Most of the sparse matrix examples in these notes are derived from this matrix. In C the matrix can be stored in a two-dimensional array, `int a[6][6]`. This array reserves 36 cells for 8 meaningful numbers. A 1000×1000 matrix with only five nonzero entries in each row takes 4 MB with four-byte integers; the nonzero entries number only 5,000.

	0	1	2	3	4	5
0	15	0	0	22	0	-15
1	0	11	3	0	0	0
2	0	0	0	-6	0	0
3	0	0	0	0	0	0
4	91	0	0	0	0	0
5	0	0	28	0	0	0

Figure 5.7: A *6 × 6* sparse matrix. Only 8 of the 36 cells are nonzero (dark cells).

Wasted memory is not the only problem. In a two-dimensional array, operations such as transpose, addition and multiplication walk through all cells. The transpose of a *1000 × 1000* matrix copies a million cells, although only 5,000 entries change place. The aim of sparse matrix representations is to tie both memory and running time to the number of nonzero entries.

5.6.2 The triple representation

Each nonzero entry is fully determined by three pieces of information: its row, its column and its value. Gathering these three pieces in one structure is the two-dimensional counterpart of the ⟨exponent, coefficient⟩ pair of the term array.

Definition 5.3

A **triple** is a ⟨row, column, value⟩ record that holds one nonzero entry of a sparse matrix. In the **triple representation** the matrix is an array of triples. Element 0 of the array is the header: it holds the number of rows, the number of columns and the number of nonzero entries. The triples are ordered by row, and within the same row by increasing column.

```
#define MAX_TERMS 101
#define MAX_COL 50   /* max columns */
typedef struct {
  int row, col, value;   /* row, column, value */
} term;
term M[MAX_TERMS];
```

The `term` structure has three `int` fields and occupies 12 bytes. `M[0]` is the header: `M[0].row` holds the number of rows, `M[0].col` the number of columns, and `M[0].value` the number of nonzero entries. The nonzero entries are stored from `M[1]` to `M[M[0].value]`. `MAX_COL` is the size

of the helper arrays in the transpose functions; the number of columns of the matrix cannot exceed this value. Storing the header as a triple of the same type is a small trick: no separate header structure is needed, and the matrix is passed to functions as a single array.

	row	col	value
M[0]	6	6	8
M[1]	0	0	15
M[2]	0	3	22
M[3]	0	5	-15
M[4]	1	1	11
M[5]	1	2	3
M[6]	2	3	-6
M[7]	4	0	91
M[8]	5	2	28

Figure 5.8: The triple table of the matrix in Figure 5.7. `M[0] = (6, 6, 8)` is the header; the eight triples are ordered by row, then by column.

The ordering condition is row-major order applied to triples. A two-dimensional array sits in memory row by row; the triple table reads the nonzero entries in the same order. In Figure 5.8 the three entries of row 0 (columns 0, 3, 5) come first, then the two entries of row 1 (columns 1, 2). Row 3 has no nonzero entry, so the table has no triple with row number 3. Thanks to this order, the entries of a row are consecutive in the table, and reading a row means reading a range of the table.

The book by Horowitz defines the sparse matrix abstract data type with four operations: `Create(max_row, max_col)` constructs an empty matrix, `Transpose(a)` returns the transpose, and `Add(a, b)` and `Multiply(a, b)` return the sum and the product. The triple representation is only one implementation of these operations; there are others, such as linked lists and compressed sparse row storage.

5.6.3 When does the triple representation pay off?

A two-dimensional array stores *rows · cols* integers. The triple representation stores *t + 1* triples including the header, that is, *3(t + 1)* integers. For the triple representation to take less space, we need

> 3(t + 1) < rows · cols, that is, t < rows · cols / 3 − 1

Roughly, if less than a third of the matrix is nonzero, the triple representation pays off. When a third is nonzero, the two representations cost about the same memory; if the matrix is denser, the triple representation takes more space and is not preferred.

Example 5.4

For the matrix in Figure 5.7: $t = 8$; the two-dimensional array takes 36 integers and the triple table takes $3 \cdot 9 = 27$. The triple representation wins. The threshold is $3(t + 1) < 36$, that is, $t \leq 10$; at $t = 11$ the two representations tie at 36 integers each.

In a 5×5 matrix the array takes 25 integers. The condition $3(t + 1) > 25$ holds for $t \geq 8$: at $t = 7$ the triples take 24 integers, at $t = 8$ they take 27. With eight or more nonzero entries the triple representation takes more space.

This calculation compares only memory. For time the picture is different: in the triple representation an entry cannot be reached directly like `a[i][j]`. To read the entry (i, j), the table must be searched. Thanks to sortedness, binary search finds it in $O(\log t)$, but that is still not $O(1)$. A sparse representation suits operations that walk through all nonzero entries rather than reading single entries. Transpose and multiplication are such operations.

Common mistake

Mistaking `M[0]` for a nonzero entry. Loops start at `j = 1` and end with the condition `j <= M[0].value`. A loop that starts at `j = 0` processes the header too: the header (6, 6, 8) of the 6 × 6 matrix would be read as "the value at row 6, column 6 is 8". There is no such cell in the matrix, and it can lead to writing outside the bounds of the array.

5.7 Transposing

The transpose is one of the simplest matrix operations, and it shows both the strength and the trap of the sparse representation.

5.7.1 The transpose and the ordering problem

Definition 5.4

The **transpose** M' of a matrix M is the matrix whose rows are the columns of M: $M'[j][i] = M[i][j]$ for every i, j. The transpose of a matrix of size $rows \times cols$ has size $cols \times rows$.

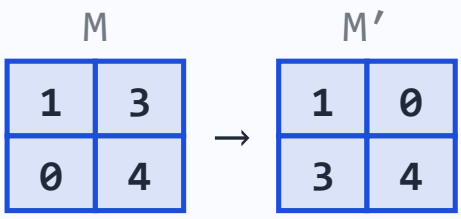


Figure 5.9: A *2 × 2* matrix and its transpose. The first row (1, 3) becomes the first column of the transpose.

On a two-dimensional array the transpose is two nested loops: `t[j][i] = m[i][j]`. With triples, the first idea that comes to mind is to swap the `row` and `col` fields of every triple. This gives the right position of each entry, but breaks the order. In the matrix of Figure 5.7, `M[1] = (0, 0, 15)`, `M[2] = (0, 3, 22)` and `M[3] = (0, 5, −15)`. Swapping the fields gives (0, 0, 15), (3, 0, 22), (5, 0, −15). The next one, `M[4] = (1, 1, 11)`, becomes (1, 1, 11). The result table runs 0, 3, 5, 1, … and is no longer in row order. Every algorithm that relies on row order, such as multiplication and binary search, would work incorrectly with this table.

There are two crude ways to restore the order. The fields are swapped and the table is sorted afterwards; or each triple is inserted at the right place in the result table and the following triples are shifted by one position. Both are expensive. The course instructor puts it this way: if shifting and moving are used, the data structure is not being given its due. A better solution comes from the definition of the transpose. The rows of the transpose are the columns of *M*. So if *M* is scanned column by column, the rows of the transpose are produced in order.

5.7.2 transpose(): scanning column by column

```
void transpose(term M[], term T[]) {
  int n = M[0].value, i, j, cur;
  T[0].row = M[0].col;  T[0].col = M[0].row;
  T[0].value = n;
  if (n > 0) {
    cur = 1;                        /* next slot in T */
    for (i = 0; i < M[0].col; i++)     /* each column */
      for (j = 1; j <= n; j++)         /* each term */
        if (M[j].col == i) {
          T[cur].row = M[j].col; T[cur].col = M[j].row;
          T[cur].value = M[j].value;  cur++;
        }
  }
}
```

The function first writes the header: the number of rows of the transpose is the number of columns of *M*, and its number of columns is the number of rows of *M*. The number of nonzero entries does not change. If the matrix is not empty, two nested loops run. The outer loop makes one round for each

column `i` of *M*. The inner loop scans all triples of *M* and writes those whose column is `i` into `T`. `cur` is the next free slot in `T`.

Why the result comes out sorted rests on two observations. First, since the outer loop walks through the columns in increasing order, the entries of column 0 of *M* are written to `T` first, then those of column 1. These are rows 0 and 1 of `T`; row order is satisfied. Second, since the entries of the same column stand in row order in *M*, the inner loop finds them in row order. In `T` these row numbers become column numbers; so the column order within the same row is satisfied as well.

Note that the inner loop starts at `j = 1`. If it started at `j = 0`, the header `M[0]` would be scanned too. `M[0].col` is the number of columns and never equals any value of `i`, because `i` goes up to at most one less than the number of columns. The result would therefore not be corrupted, but an unnecessary comparison would be made for every column. The correct start is `j = 1`.

5.7.3 Code trace and analysis

A small matrix is used for the trace: *M*, of size *3 × 4* with four nonzero entries. The nonzero entries are `M[1] = (0, 0, 7)`, `M[2] = (0, 3, 3)`, `M[3] = (1, 1, 4)` and `M[4] = (2, 0, 2)`. The columns of the triples are 0, 3, 1, 0 respectively.

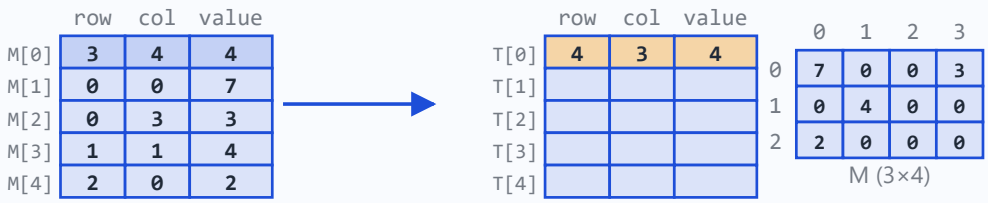


Figure 5.10: The start of `transpose`. On the left the triples of *M*, in the middle `T` with its header written, on the right the two-dimensional view of *M*.

`i`	**Comparison with** `M[1..4].col`	**Matching** `j`	**Written**	`cur` **(after)**
—	header	—	`T[0] = (4, 3, 4)`	1
0	0 = 0, 3 ≠ 0, 1 ≠ 0, 0 = 0	1, 4	`T[1] = (0, 0, 7)`, `T[2] = (0, 2, 2)`	3
1	0 ≠ 1, 3 ≠ 1, 1 = 1, 0 ≠ 1	3	`T[3] = (1, 1, 4)`	4
2	0 ≠ 2, 3 ≠ 2, 1 ≠ 2, 0 ≠ 2	—	—	4
3	0 ≠ 3, 3 = 3, 1 ≠ 3, 0 ≠ 3	2	`T[4] = (3, 0, 3)`	5

Table 5.7: Trace of `transpose(M, T)`, summarized per column.

The outer loop runs four rounds and the inner loop four steps per round: *4 · 4 = 16* comparisons are made. Only four of them match; twelve are wasted. Column 2 has no nonzero entry at all, yet all triples are still scanned. The result `T` comes out sorted by row automatically: (0, 0, 7), (0, 2, 2), (1, 1, 4), (3, 0, 3).

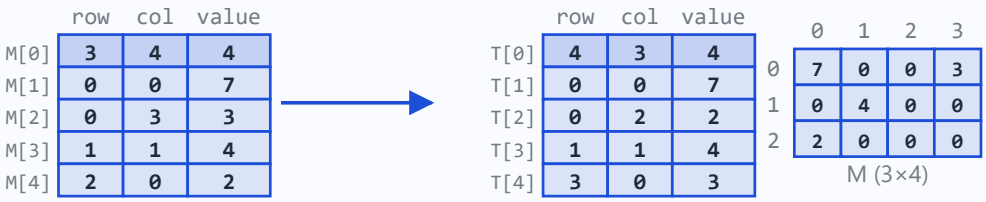


Figure 5.11: `T` when `transpose` finishes. The transpose has size *4 × 3*; the triples are in row order, then column order.

The function has also been applied to the *6 × 6* matrix of Figure 5.7. The result table is (0, 0, 15), (0, 4, 91), (1, 1, 11), (2, 1, 3), (2, 5, 28), (3, 0, 22), (3, 2, −6), (5, 0, −15) (Figure 5.12). Here $6 \cdot 8 = 48$ comparisons are made.

	row	col	value
T[0]	6	6	8
T[1]	0	0	15
T[2]	0	4	91
T[3]	1	1	11
T[4]	2	1	3
T[5]	2	5	28
T[6]	3	0	22
T[7]	3	2	-6
T[8]	5	0	-15

Figure 5.12: The transpose `T` of the *6 × 6* example matrix. The 91 of *M* (row 4, column 0) sits at row 0, column 4 in the transpose.

The complexity can be read off the nested loops. The outer loop runs *cols* times and the inner loop *terms* times; the inner body does constant work. The time is $O(cols \cdot terms)$. This cost depends on how sparse the matrix is. If the matrix is full, $terms = rows \cdot cols$ and the cost rises to $O(cols^2 \cdot rows)$. On a two-dimensional array the transpose takes $O(rows \cdot cols)$. On a full matrix, `transpose` is slower by a factor of *cols*.

So the triple representation has gained in memory but lost in time. This trade is meaningful only when $terms \ll rows \cdot cols$. The source of the problem is clear: the same array *M* is rescanned from start to end for every column. If it were known beforehand where each term will go in `T`, *M* could be scanned once and every term placed directly. For example, suppose it is known that row 0 of the transpose will have 2 entries and row 1 will have 3. Row 0 starts at `T[1]`, row 1 at `T[3]`, row 2 at `T[6]`. The fast transpose computes this information in advance.

5.7.4 fast_transpose(): count first, then place

Definition 5.5

The **fast transpose** is a method that transposes a sparse matrix in two passes. The first pass counts the nonzero entries in each column and computes the position in the table at which each row of the transpose will start. The second pass writes each triple directly to that position.

```
void fast_transpose(term M[], term T[]) {
  int i, j, nc = M[0].col, nt = M[0].value;
  int row_terms[MAX_COL], start_pos[MAX_COL];
  T[0].row = nc;  T[0].col = M[0].row;  T[0].value = nt;
  if (nt <= 0) return;                    /* empty matrix */
  for (i = 0; i < nc; i++) row_terms[i] = 0;
  for (i = 1; i <= nt; i++) row_terms[M[i].col]++;
  start_pos[0] = 1;
  for (i = 1; i < nc; i++)
    start_pos[i] = start_pos[i-1] + row_terms[i-1];
  for (i = 1; i <= nt; i++) {
    j = start_pos[M[i].col]++;       /* take the slot, advance */
    T[j].row = M[i].col;  T[j].col = M[i].row;
    T[j].value = M[i].value;
  }
}
```

The function uses two helper arrays. `row_terms[i]` is the number of entries in row `i` of the transpose, that is, in column `i` of *M*. `start_pos[i]` is the first position of row `i` of the transpose in `T`. The code has four loops:

1. `row_terms` **is zeroed.** In C, local arrays are not initialized; the values in them are indeterminate. If the zeroing is skipped, the counters start from random values.
2. **Counting.** For each triple of *M*, the counter of its own column is incremented: `row_terms[M[i].col]++`.
3. **Starting positions.** Row 0 starts at `T[1]`. Each row is found by adding the number of entries of the previous row to the start of the previous row: `start_pos[i] = start_pos[i-1] + row_terms[i-1]`. This is a **prefix sum**: `start_pos[i]` is 1 plus the sum of all counters from 0 to *i* − *1*.
4. **Placement.** *M* is scanned once from start to end. Each triple is written where the `start_pos` value of its column points, and this value is incremented. The next triple from the same column falls in the next position.

The line `j = start_pos[M[i].col]++;` of the fourth loop is a dense expression. First `M[i].col` is read; this is the index into the array. The postfix `++` operator first gives the old value of `start_pos[M[i].col]` to `j`, then increments the array cell. If the prefix form `++start_pos[M[i].col]` were written, the increment would happen first and `j` would be one too large: the first triple would be written to `T[2]`, and the triple belonging to the last position would overflow the array. The instructor's warning is about this line: a programmer must know well the operator precedence of the language they write in. Here the square brackets and the `.` operator are applied before the postfix `++`; what is incremented is that cell of the `start_pos` array, not `M[i].col`.

Why is the order of the triples preserved? Since M is scanned in row order, the entries of the same column arrive in row order. Each is written to the next free position of its own row. Within each row of the transpose, the entries are in increasing order of the row numbers in M, that is, of the column numbers in the transpose. The order of the rows themselves comes from the `start_pos` values being increasing.

5.7.5 Code trace: the two phases

The same 3×4 matrix is used. The first phase counts and finds the starting positions (Table 5.8).

Step	Action	`row_terms`	`start_pos`
1	header: `T[0] = (4, 3, 4)`; counters zeroed	[0, 0, 0, 0]	—
2	`M[1]` is in column 0	[1, 0, 0, 0]	—
3	`M[2]` is in column 3	[1, 0, 0, 1]	—
4	`M[3]` is in column 1	[1, 1, 0, 1]	—
5	`M[4]` is in column 0	[2, 1, 0, 1]	—
6	`start_pos[0] = 1`	[2, 1, 0, 1]	[1, ·, ·, ·]
7	`start_pos[1] = 1 + 2 = 3`	[2, 1, 0, 1]	[1, 3, ·, ·]
8	`start_pos[2] = 3 + 1 = 4`	[2, 1, 0, 1]	[1, 3, 4, ·]
9	`start_pos[3] = 4 + 0 = 4`	[2, 1, 0, 1]	[1, 3, 4, 4]

Table 5.8: `fast_transpose` first phase: counting and prefix sum.

The equality in the last step stands out: `start_pos[2]` and `start_pos[3]` are both 4. Since column 2 has no entries, row 2 of the transpose is empty and row 3 starts where row 2 would have started. An empty row shows up as two consecutive equal starting positions.

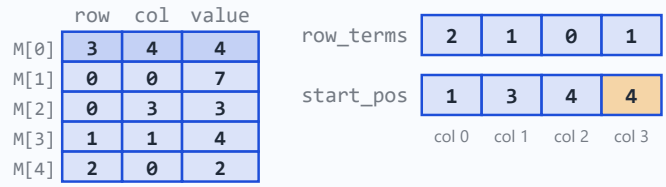


Figure 5.13: The end of the first phase: `row_terms = [2, 1, 0, 1]`, `start_pos = [1, 3, 4, 4]`.

The second phase scans *M* once (Table 5.9).

`i`	`M[i]`	`j = start_pos[M[i].col]++`	**Written**	`start_pos` **(after)**
1	(0, 0, 7)	`start_pos[0]` : `j = 1`	`T[1] = (0, 0, 7)`	[2, 3, 4, 4]
2	(0, 3, 3)	`start_pos[3]` : `j = 4`	`T[4] = (3, 0, 3)`	[2, 3, 4, 5]
3	(1, 1, 4)	`start_pos[1]` : `j = 3`	`T[3] = (1, 1, 4)`	[2, 4, 4, 5]
4	(2, 0, 2)	`start_pos[0]` : `j = 2`	`T[2] = (0, 2, 2)`	[3, 4, 4, 5]

Table 5.9: `fast_transpose` second phase: placement.

The triples are written to `T` out of order: first `T[1]`, then `T[4]`, `T[3]` and `T[2]`. But each goes to the right place, and in the end `T` is the same as the table produced by `transpose`. When the phase ends, `start_pos[i]` points to the position after the last entry of row `i`; that is, `start_pos[i]` is now the start of row `i + 1`. Because column 2 is empty, `start_pos[2]` has never changed.

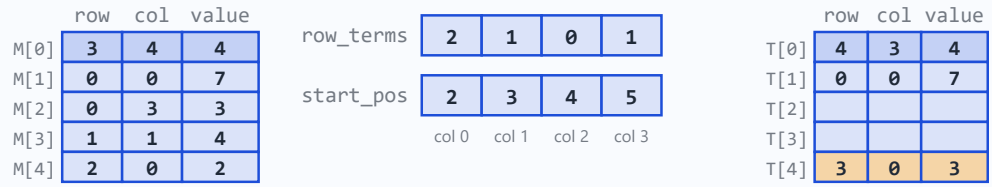


Figure 5.14: The second step of the second phase: `M[2] = (0, 3, 3)` has been written directly to `T[4]`; `T[2]` and `T[3]` are still empty.

On the *6 × 6* example matrix the first phase produces `row_terms = [2, 1, 2, 2, 0, 1]` and `start_pos = [1, 3, 4, 6, 8, 8]`. Since column 4 is empty, `start_pos[4]` and `start_pos[5]` are equal. These values have been verified by a program.

5.7.6 Analysis, the one-array version and measurement

The four loops of `fast_transpose` run *cols*, *terms*, *cols* − *1* and *terms* rounds respectively. None is nested. Together with the constant number of operations outside the loops, the total work is

> $2 \cdot \text{terms} + 2 \cdot \text{cols} + \text{constant} = O(\text{cols} + \text{terms})$

On a full matrix *terms = rows · cols*, so the cost is *O(rows · cols)*: the same order as the transpose of a two-dimensional array. On a sparse matrix the cost is proportional to the number of nonzero entries. The price is two helper arrays of size *cols*: *O(cols)* extra memory.

Theorem 5.2 (Cost of the two transposes)

For a sparse matrix of size *rows × cols* with *t* nonzero entries, `transpose` runs in $\Theta(cols \cdot t)$ time and `fast_transpose` in $\Theta(cols + t)$ time. Both produce a transpose in the correct order.

The fast transpose can also be written with a single helper array. This recalls the instructor's question: it was said that fast transpose can be done with one array; here it is. Once the counting is finished, the counters are no longer needed; the same array is converted in place into the starting positions:

```
/* convert row_terms in place into start_pos */
for (i = 0, pos = 1; i < nc; i++) {
  int c = row_terms[i];  /* terms in this column */
  row_terms[i] = pos;  /* now a start position */
  pos += c;
}
for (i = 1; i <= nt; i++) {
  j = row_terms[M[i].col]++;
  T[j].row = M[i].col;  T[j].col = M[i].row;
  T[j].value = M[i].value;
}
```

In this code `row_terms` is the array of counts, as in the earlier version. The first loop reads each counter, writes the starting position of that row in its place, and advances `pos` by the counter. If the old value of the counter is not saved in the local variable `c`, the overwritten value is read and `pos` advances wrongly. The second loop is the same as the placement loop of the earlier version. This form was compiled as a complete function in the test program of these notes and was seen to give the same result as the version with two helper arrays.

The one-array version is counting sort itself. Counting sort sorts items whose keys come from a small range: first it counts how many items there are of each key, finds the starting position of each key with a prefix sum, then puts the items in their places one by one. In the fast transpose, the key is the column number. Because the relative order of items with the same key in the input is preserved, the sort is stable and the row order is not disturbed.

The difference is striking on large matrices. Table 5.10 gives measurements of the two functions of the slides, compiled with `-O2` after only enlarging the constants `MAX_TERMS` and `MAX_COL`. Each row has ten nonzero entries. The times vary by machine; what matters is the growth rate.

Size	Nonzero entries	`transpose`	`fast_transpose`
1000 × 1000	10,000	≈ 4	≈ 0.04
2000 × 2000	20,000	≈ 14	≈ 0.08
4000 × 4000	40,000	≈ 60	≈ 0.26
8000 × 8000	80,000	≈ 254	≈ 0.36

Table 5.10: Transpose times (milliseconds; they vary by machine).

When the size doubles, both the number of columns and the number of nonzero entries double. Since the time of `transpose` is proportional to *cols* · *t*, it roughly quadruples: 4, 14, 60, 254. The time of `fast_transpose` is proportional to *cols* + *t* and roughly doubles. On the largest matrix the difference between the two methods is close to a factor of seven hundred. The small times are near the limit of measurement precision; that is why the fast transpose was run fifty times and averaged.

Common mistake

Swapping only `row` and `col` in the transpose and forgetting to sort the table. The result contains the right entries but is not in row order. A multiplication done with this table gives a wrong result, because it assumes that the entries of the same row are consecutive. The bug may not show up on small examples: a one-row or diagonal matrix stays sorted even when the fields are simply swapped.

5.8 Sparse matrix multiplication

Matrix multiplication is the operation where the sparse representation helps most and where the code must be written most carefully. The code in this section is a simplified adaptation of Horowitz's `mmult` program.

5.8.1 Row times column

Let A be an $m \times n$ matrix and B an $n \times p$ matrix. Their product $D = A \cdot B$ has size $m \times p$, and each of its entries is given by

$$d_{ij} = \Sigma_{k=0}^{n-1} a_{ik} \cdot b_{kj}$$

Row i of A and column j of B are multiplied entry by entry and summed. In sparse matrices most terms of this sum are zero. The product $a_{ik} \cdot b_{kj}$ is nonzero only if both entries are nonzero. So only the pairs of entries that are nonzero for the same value of k need to be multiplied.

The product of two sparse matrices need not be sparse. In each of the two matrices of Figure 5.15, three of the nine cells are nonzero, yet all nine cells of their product are nonzero. All nonzero entries of A are in column 0, and all nonzero entries of B are in row 0; every pair (i, j) meets at $k = 0$.

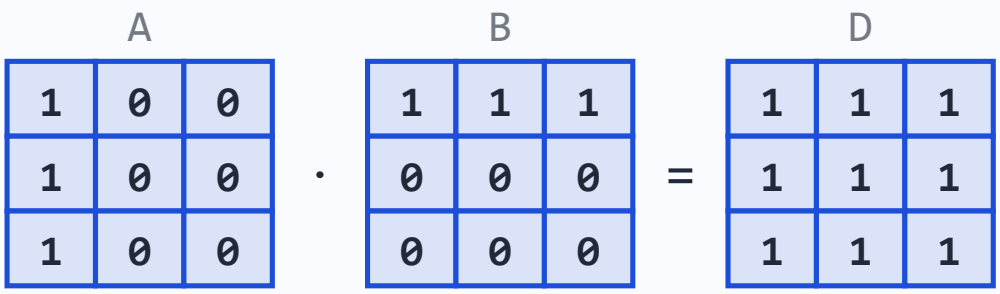


Figure 5.15: The product of two sparse matrices can be full. Three of the nine cells are nonzero in A and in B; all nine cells of D are 1.

In the triple representation, row i of A is consecutive. But column j of B is scattered over the table: at most one entry from each row, at different places in the table. Summing one column requires scanning the whole table. The solution is to transpose B first. In B^T, obtained with `fast_transpose`, the columns of B become consecutive rows. Then each row of A and each row of B^T are merged by their `col` fields and multiplied. The column of an entry in row i of A is k; the column of an entry in row j of B^T is its row in B, which is again k. The entries of the two rows with the same `col` value are the pairs to be multiplied.

Which matrix is transposed matters. It would be wrong to say “the first matrix is transposed first”. In Horowitz's `mmult`, and logically, the matrix that is transposed is the second one, B. The rows of A are already consecutive; the problem lies in the columns of B.

5.8.2 Helpers: row_end(), dot(), store()

The product is written with three small helpers. The first two are these:

```
/* where the row of the term at s ends */
int row_end(term t[], int s) {
  int e = s;
  while (e <= t[0].value &&
         t[e].row == t[s].row) e++;
  return e;
}
/* row a[x..ex) times row b[y..ey) */
int dot(term a[], int x, int ex,
        term b[], int y, int ey) {
  int sum = 0;
  while (x < ex && y < ey)
    if (a[x].col < b[y].col) x++;
    else if (a[x].col > b[y].col) y++;
    else sum += a[x++].value * b[y++].value;
  return sum;
}
```

`row_end(t, s)` finds where the triples in the same row as the triple `t[s]` end. `e` starts at `s` and advances until the row number changes or the table ends. The returned `e` is the index one past the last entry of the row. The row is the half-open range *[s, e)*: `s` included, `e` excluded. This is the usual form of array loops in C; an empty range appears as $s = e$ and the length of the range is $e - s$. The order in the condition matters: `e <= t[0].value` is tested first. Because the `&&` operator short-circuits, `t[e]` is never read with an `e` past the end of the table.

`dot` computes the dot product of the row *[x, ex)* of `a` and the row *[y, ey)* of `b`. This is a two-pointer merge like `padd`. The entries of both rows are sorted by column. If the columns differ, the pointer with the smaller column advances, because that column has no counterpart in the other row. If the columns are equal, the two values are multiplied and added to the sum, and both pointers advance. The loop stops when one of the rows is exhausted; the remaining entries cannot have a counterpart. The cost is proportional to the sum of the numbers of entries of the two rows.

The third helper writes the result:

```
void store(term d[], int *nd,
           int r, int c, int s) {
  if (s == 0) return;
  if (++*nd >= MAX_TERMS) {
    fprintf(stderr, "Too many terms\n");
    exit(1);
  }
  d[*nd].row = r;
  d[*nd].col = c;
  d[*nd].value = s;
}
```

`store` writes nothing if the sum is 0; zero entries do not enter the sparse table. Otherwise the counter is incremented first and then the triple is written. In the expression `++*nd`, the prefix `++` and `*` have the same precedence and bind right to left: it means `++(*nd)`. First the counter that `nd` points to is incremented, then the new value is compared with the limit. The result exceeding `MAX_TERMS` is an error. Since there is no guarantee that the product stays sparse, this check is not decoration.

5.8.3 smult() and a code trace

```
void smult(term a[], term bt[], term d[]) {
  int ra, ea, rb, eb, nd = 0;
  for (ra = 1; ra <= a[0].value; ra = ea) {
    ea = row_end(a, ra);       /* row of A [ra,ea) */
    for (rb = 1; rb <= bt[0].value; rb = eb) {
      eb = row_end(bt, rb);    /* column of B [rb,eb) */
      store(d, &nd, a[ra].row, bt[rb].row,
            dot(a, ra, ea, bt, rb, eb));
    }
  }
  d[0].row = a[0].row;  d[0].col = bt[0].row;
  d[0].value = nd;
}
```

The function takes A and `bt`, the transpose of B. The outer loop walks through the rows of A: `ra` is the first triple of the row and `ea = row_end(a, ra)` is the end of the row. The next round starts with `ra = ea`, that is, from the next row. The inner loop walks the same way through the rows of `bt`, that is, the columns of B. For each pair of rows, `dot` computes the dot product and `store` writes it to the result if it is not zero. The row of the result triple is `a[ra].row` and its column is `bt[rb].row`: the row number in B^T is the column number in B.

The result triples come out in the right order automatically. The outer loop walks the rows of A and the inner loop the columns of B in increasing order. This is row-major order itself. The header is written last, because the number of nonzero entries is known only when the loops are finished. Empty rows and empty columns are never visited; the loops walk only through the rows that appear in the table.

The trace uses $A = [[1, 0, 2], [0, 3, 0]]$ (2×3) and $B = [[4, 0], [0, 5], [6, 0]]$ (3×2). The tables are: `a` = (2, 3, 3) | (0, 0, 1), (0, 2, 2), (1, 1, 3). The transpose of B is 2×3: `bt` = (2, 3, 3) | (0, 0, 4), (0, 2, 6), (1, 1, 5).

Step	Row of A	Column of B	`dot`	`store`	`nd`
1	0: `a[1..2]` (columns 0, 2)	0: `bt[1..2]` (columns 0, 2)	$1 \cdot 4 + 2 \cdot 6 = 16$	`d[1] = (0, 0, 16)`	1
2	0: `a[1..2]`	1: `bt[3..3]` (column 1)	no common column: 0	not stored	1
3	1: `a[3..3]` (column 1)	0: `bt[1..2]`	no common column: 0	not stored	1
4	1: `a[3..3]`	1: `bt[3..3]`	$3 \cdot 5 = 15$	`d[2] = (1, 1, 15)`	2

Step	Row of *A*	Column of *B*	`dot`	`store`	`nd`
5	header: `d[0] = (2, 2, 2)`				2

Table 5.11: Trace of `smult(a, bt, d)`.

In the first step `dot` merges the columns {0, 2} of row 0 of *A* with the rows {0, 2} of column 0 of *B*. There are two matches: *1 · 4* at *k = 0* and *2 · 6* at *k = 2*. The sum is 16. In the second step the columns of *A* are {0, 2} and the rows of column 1 of *B* are {1}; there is no common value, and since the sum is 0 nothing is written. The result is *D = [[16, 0], [0, 15]]*.

A (2×3)

	row	col	value
a[0]	2	3	3
a[1]	0	0	1
a[2]	0	2	2
a[3]	1	1	3

Bᵀ (B: 3×2)

	row	col	value
bt[0]	2	3	3
bt[1]	0	0	4
bt[2]	0	2	6
bt[3]	1	1	5

D = A·B (2×2)

	row	col	value
d[0]	2	2	2
d[1]	0	0	16
d[2]	1	1	15

Figure 5.16: When `smult` finishes: the triple tables of *A*, B^T and the result *D*. The header of *D* is written last: *2 × 2*, two nonzero entries.

The correctness of the code has been tested beyond small examples. The test program generated 3,000 pairs of matrices with sizes chosen at random between 1 and 6 and about a quarter of the entries nonzero. The result of `fast_transpose` + `smult` for each pair came out equal to the classical product computed with three nested loops. In the same test, `transpose`, `fast_transpose` and the one-array version also gave the same transpose for every matrix. The example in Figure 5.15 shows that the product can produce a full table of nine entries.

5.8.4 Complexity

Let the number of nonzero entries of *A* be t_A and that of *B* be t_B. The cost has three parts:

- **Transpose.** `fast_transpose(B)` : $O(cols_B + t_B)$.
- **Walking through B^T.** For each row of *A*, `bt` is walked from start to end; the calls to `row_end` and the steps of `dot` on the `bt` side add up to $O(t_B)$ per row. For at most $rows_A$ rows of *A*: $\Sigma_i\, t_B = rows_A \cdot t_B$.
- **Walking through the rows of *A*.** For each pair *(i, j)*, `dot` also walks through row *i* of *A*. *j* takes at most $cols_B$ values; for each *j* the rows of *A* total t_A entries: $\Sigma_j\, t_A = cols_B \cdot t_A$.

Theorem 5.3 (Cost of sparse multiplication)

The product of matrices *A* (t_A nonzero entries) and *B* (t_B nonzero entries) given in triple representation is computed, transpose included, in $O(cols_B \cdot t_A + rows_A \cdot t_B)$ time.

The cost $O(cols_B + t_B)$ of the transpose stays inside this sum. The classical product takes $O(rows_A \cdot cols_A \cdot cols_B)$: three nested loops. If the matrices are full, $t_A = rows_A \cdot cols_A$ and $t_B = cols_A \cdot cols_B$, and the sparse version also rises to the order of $O(rows_A \cdot cols_A \cdot cols_B)$. The order is the same, but the constant is larger: three fields are read for every entry and comparisons are made.

The sparse version pays off when t_A and t_B are a small fraction of the matrix sizes. For example, let two matrices of size 1000×1000 have five nonzero entries in each row: $t_A = t_B = 5000$. The classical product makes 10^9 multiplications. The bound of the sparse product is $1000 \cdot 5000 + 1000 \cdot 5000 = 10^7$; a hundred times fewer. But there is no guarantee that the result D will be sparse. A full result as in Figure 5.15 can exceed `MAX_TERMS`; this is why the check in `store` exists.

Common mistake

Transposing A in the product. The rows of A are already consecutive; transposing would scatter them and does not solve the problem. The matrix that must be transposed is B, whose columns are scattered. Another mistake is to assume that the result stays sparse and allocate a small result table.

5.9 Other representations

The triple representation is instructive and simple, but it is not the only choice. The format most used in scientific computing is compressed sparse row storage. In applications that need frequent insertion and deletion, a linked representation comes to the fore.

5.9.1 Compressed sparse row (CSR)

Definition 5.6

Compressed sparse row (CSR) storage keeps a sparse matrix in three arrays. `val` holds the values of the nonzero entries and `col_ind` holds their column numbers, both in row-major order; both have length t. The array `row_ptr` has length $rows + 1$: `row_ptr[i]` is the index in `val` of the first entry of row i, and `row_ptr[rows] = t`.

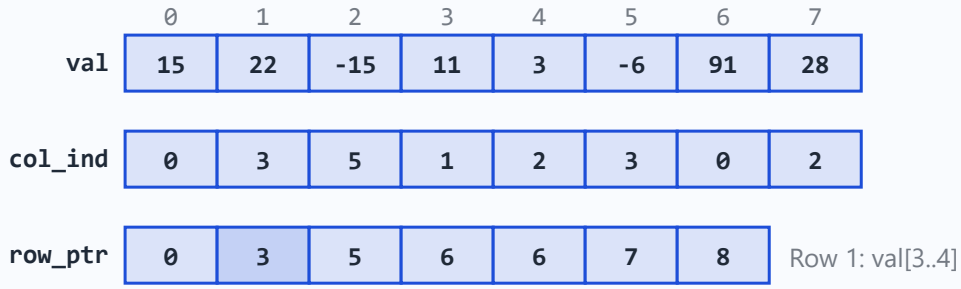


Figure 5.17: The CSR representation of the *6 × 6* example matrix. Row 1 is `val[3..4]`; row 3, which is empty, shows up as two consecutive equal `row_ptr` values (6, 6).

In the triple representation the row number of every entry is written out separately. Since entries of the same row are consecutive, this information is repeated needlessly. CSR drops the row numbers and keeps where each row starts. The entries of row *i* are in the range `val[row_ptr[i] .. row_ptr[i+1] - 1]`; the length of the range is `row_ptr[i+1] - row_ptr[i]`.

In the *6 × 6* example, `row_ptr = [0, 3, 5, 6, 6, 7, 8]`. Row 0 is `val[0..2]`: 15, 22, −15. Row 3 has no entries; `row_ptr[3] = row_ptr[4] = 6`. Memory: `val` 8, `col_ind` 8, `row_ptr` 7 integers, 23 in total. The triple representation took 27 integers. The difference becomes noticeable in large matrices: triples take *3t* integers, CSR takes *2t + rows + 1*.

The most important use of CSR is matrix–vector multiplication: *y = M · x*. Iterative equation solvers do this operation thousands of times.

```
/* y = M * x  (matrix-vector product) */
for (i = 0; i < rows; i++) {
  y[i] = 0;
  for (k = row_ptr[i]; k < row_ptr[i+1]; k++)
    y[i] += val[k] * x[col_ind[k]];
}
```

The outer loop walks through the rows. The inner loop walks through the nonzero entries of row *i*: the value of each entry is multiplied by the vector entry in the same column and added to `y[i]`. The expression `x[col_ind[k]]` is an indirect access: first the column number is read, then that position of the vector. The outer loop runs *rows* rounds and the inner loops *t* rounds in total. The cost is *O(rows + t)*. When the example matrix is multiplied by *x = (1, 2, 3, 4, 5, 6)*, the result is *y = (13, 31, −24, 0, 91, 84)*; the first entry is *15 · 1 + 22 · 4 − 15 · 6 = 13*.

`row_ptr` is the same idea as `start_pos` in `fast_transpose`: the starting position of each row is the prefix sum of the entry counts of the earlier rows. Building CSR from a triple table is also done by counting: the entries of each row are counted and the prefix sum is taken. The column-major counterpart is **CSC** (compressed sparse column): columns are compressed instead of rows. The CSR representation of a matrix consists of the same arrays as the CSC representation of its transpose.

5.9.2 The linked sparse matrix

The triple array and CSR are very good for a fixed matrix, but struggle when the matrix changes. Inserting or deleting an entry requires shifting the sorted array and takes $O(t)$. The triple array also has the limit `MAX_TERMS`.

In the linked representation every nonzero entry is a node. A node holds the row, the column, the value and two links: to the next entry in the same row and to the next entry in the same column. The nodes in the same row form a **circular linked list**; so do those in the same column. Each row and each column has a **header node**. The header nodes are also linked among themselves, so the whole matrix can be traversed from a single entry point. Horowitz builds this structure in detail in the linked lists chapter of his book.

The gain of this representation is flexibility. Once the position has been found, inserting or deleting an entry takes a few link changes, in $O(1)$ time. Traversal by both row and column is easy: walking along a row or a column means walking along the corresponding circular list. So multiplication does not need a transpose. The price is the extra memory of two pointers per entry; on a 64-bit system 16 bytes are added to the 12 bytes of three integers.

The linked implementation of a sparse matrix is a direct application of linked lists. Linking a node into two lists at once, updating the links of both lists consistently in insertion and deletion, and freeing all nodes without a leak are a good test of linked-list skill. The course instructor stresses in particular that this work must be each student's own effort: if two homework papers are laid on top of each other and held up to the light, they should not look the same.

5.10 Complexity summary

Table 5.12 brings together the costs of the operations studied in these notes.

Operation	Time	Remark
Addition with coefficient arrays	$O(d)$	d: degree; wasteful for sparse polynomials
`padd` (n, m terms)	$\Theta(n + m)$	sorted terms + merging
`pmult` (array, repeated `padd`)	$O(n^2 m)$	smaller polynomial outside: $O(min^2 \cdot max)$; leftovers pile up
`pmult_list` (linked list)	$O(n^2 m)$ worst case	the search is shortened by `anchor`; depends on the arrangement in measurements
`transpose`	$O(cols \cdot terms)$	on a full matrix $O(cols^2 \cdot rows)$
`fast_transpose`	$O(cols + terms)$	$O(cols)$ extra memory; counting sort
Sparse product (`smult` / `mmult`)	$O(cols_B \cdot t_A + rows_A \cdot t_B)$	B^T first
$M \cdot x$ with CSR	$O(rows + terms)$	row-start array

Table 5.12: Costs of polynomial and sparse matrix operations.

The table has a common lesson. All the fast algorithms rest on sortedness and merging: `padd` merges two sorted term arrays, `dot` merges two sorted rows; `fast_transpose` produces a sorted table with counting sort. The memory gain of the sparse representation turns into a time gain only if this order is preserved.

5.11 Common mistakes

The mistakes met most often in the code of these notes are the following:

1. **Keeping the terms unsorted but still writing `padd` with two pointers.** Merging is correct only on sorted input. On unsorted input, two terms with the same exponent may never see each other and remain as two separate terms in the result.
2. **Appending a term whose sum is 0 to the result, or not deleting it from the list.** The `if (c)` check in `padd` and the call to `delete_next` in `place` exist for this reason. The memory of a deleted node must be returned with `free`.
3. **Truncating a `float` coefficient with an `int` parameter in `attach`; not checking for overflow of `avail`.** The truncation happens without a compiler warning. Without the overflow check, memory past the end of the array is corrupted.
4. **Swapping only the `row` and `col` fields in the transpose and forgetting to sort; mistaking the header `M[0]` for a term.** An unsorted transpose breaks operations that rely on order, such as multiplication. Loops must start at `1`.
5. **Confusing prefix and postfix in `start_pos[M[i].col]++`.** The postfix form uses the old value and then increments; with the prefix form every triple shifts by one position and the triple belonging to the last position is written outside the table.
6. **Transposing A in sparse multiplication; assuming the result stays sparse.** The matrix to transpose is B. The result can be full; the limit of the result table must be checked.
7. **Mixing up the order of deletion in a linked list.** If the link is updated without saving the node's address, memory leaks; if the node is freed before the link is updated, a dangling pointer remains.
8. **Not zeroing local counter arrays.** Local arrays like `row_terms` start with indeterminate values in C; they must be zeroed before the counting loop.

5.12 Summary

- **The representation matters.** The coefficient array is simple and fast for dense, low-degree polynomials, but for sparse polynomials it spends memory and time on zeros. The term array stores only the nonzero terms, sorted from the largest exponent to the smallest.
- **Sortedness and merging.** `padd` adds two sorted term arrays by walking through them with two pointers, in $\Theta(n + m)$ time. A term whose sum is 0 is not written to the result. In the shared array `P` the space of leftovers is not reclaimed; compaction or a linked list is needed.

- **Multiplication on arrays.** `pmult` multiplies by each term in turn and adds the result with `padd`: $O(n^2m)$. The smaller polynomial should go in the outer loop. Many polynomials are merged with a priority queue or two at a time.
- **The linked list.** It is flexible when the number of terms is unknown. `pmult_list` places the products in a sorted list, combines equal exponents, deletes terms that become zero and returns their memory. The variable `anchor` shortens the search; the worst case is still $\Theta(n^2m)$.
- **Triples.** A sparse matrix is stored as ⟨row, column, value⟩ triples with the header `M[0]`. The triples are in row order, then column order. The representation pays off when $3(t + 1) < rows \cdot cols$, roughly when less than a third of the matrix is nonzero.
- **Transpose.** Swapping the fields breaks the order. `transpose` scans column by column in $O(cols \cdot t)$; `fast_transpose` counts first and then places, in $O(cols + t)$ time. The fast transpose is counting sort with the column number as the key.
- **Multiplication.** First B is transposed; then each row of A is merged and multiplied with each row of B^T: $O(cols_B \cdot t_A + rows_A \cdot t_B)$. There is no guarantee that the result stays sparse.
- **Other representations.** CSR keeps row starts instead of row numbers and performs matrix–vector multiplication in $O(rows + t)$; `row_ptr` is the same prefix-sum idea as `start_pos`. The linked sparse matrix brings insertion and deletion down to $O(1)$, but spends two pointers per entry.

5.13 Key terms

polynomial abstract data type merge priority queue memory leak dangling pointer sparse matrix triple row-major order transpose fast transpose counting sort key compressed sparse row circular linked list header node

5.14 Test Yourself

1. For $A = 5x^6 - 2x^3 + 4$ and $B = 2x^6 + 2x^3 + x - 4$, how many rounds does the main loop of `padd` run, how many calls to `attach` are made, and what is the result?

   Answer

   The loop runs four rounds: (6, 6) equal, $5 + 2 = 7$ is written; (3, 3) equal, $-2 + 2 = 0$ is not written; (0, 1) B is larger, x is written; (0, 0) equal, $4 - 4 = 0$ is not written. Both polynomials are exhausted at the same time. Two calls to `attach` are made and the result is $7x^6 + x$.

2. With `avail = 5`, $A = x^2 + 1$ and $B = x^2 + x + 1$ are multiplied with `pmult`. What is the result and

what is `avail` when it finishes? How many of the positions used hold the result?

Answer

The result is $x^4 + x^3 + 2x^2 + x + 1$ (5 terms). In the first round 3 temporary terms are written and `padd` copies them to the empty partial result (3 more terms). In the second round 3 temporary terms and the new partial result of 5 terms are written. In total $3 + 3 + 3 + 5 = 14$ positions are used; `avail` becomes 19. Of the 14 positions only 5 are the result.

3. In the product $(x^2 + x)(x - 1)$, which term does `pmult_list` delete? What is the result?

Answer

In the round $p = x^2$, x^3 and $-x^2$ are added. In the round $p = x$, $x \cdot x = +x^2$ arrives; the coefficient becomes $-1 + 1 = 0$ and the x^2 node is deleted. The last product $-x$ is appended at the end. The result is $x^3 - x$.

4. Why is the first node of the result list in `pmult_list` never deleted? Which helper's restriction does this property make harmless?

Answer

The first node is the product of the terms with the largest exponents of the two polynomials. Its exponent is larger than that of all other products and no other pair of terms can produce the same exponent; its coefficient is never updated and cannot become 0. `delete_next` can delete only the node after the given node, not the first node. Since the first node is never deleted, this restriction causes no problem.

5. In a 100×100 matrix, up to how many nonzero entries does the triple representation take less space than the two-dimensional array?

Answer

The condition is $3(t + 1) < 10{,}000$. At $t = 3332$ the triples take $3 \cdot 3333 = 9999$ integers, which is smaller; at $t = 3333$ they take 10,002 integers. The limit is $t \le 3332$, about a third of the matrix.

6. The nonzero entries of a 4×5 matrix are (0, 2, 1), (1, 0, 3), (1, 2, −1), (2, 4, 6), (3, 2, 2). What are the `row_terms` and `start_pos` arrays of `fast_transpose`? In what order are the triples of the

transpose written? How many comparisons does `transpose` make?

Answer

The columns are 2, 0, 2, 4, 2: `row_terms = [1, 0, 3, 0, 1]`, `start_pos = [1, 2, 2, 5, 5]`. The transpose is (0, 1, 3), (2, 0, 1), (2, 1, −1), (2, 3, 2), (4, 2, 6). Rows 1 and 3 are empty; that is why `start_pos[1] = start_pos[2]` and `start_pos[3] = start_pos[4]`. `transpose` makes $5 \cdot 5 = 25$ comparisons.

7. What is `row_ptr` in the CSR representation of the same 4×5 matrix? How many integers do the three arrays take in total?

Answer

`row_ptr = [0, 1, 3, 4, 5]`: row 0 has one entry, row 1 has two, rows 2 and 3 have one each. `val` and `col_ind` take five integers each and `row_ptr` takes five: 15 in total. The triple representation would take $3 \cdot 6 = 18$ integers.

8. In `fast_transpose`, what happens on the 3×4 trace matrix if `j = start_pos[M[i].col]++;` is replaced with `j = ++start_pos[M[i].col];`?

Answer

The prefix form increments first and then yields the new value. The first triple (0, 0, 7) is written to `T[2]` instead of `T[1]`. Every triple goes one position past its correct place, and the last triple (2, 0, 2) lands in `T[3]` while (0, 3, 3) lands in `T[5]`. `T[1]` stays empty. `T[5]` is beyond `T[4]`, the last valid position of a table with four nonzero entries; had the array been allocated for exactly five triples, the bound would be exceeded.

9. Two matrices of size 1000×1000 each have 3,000 nonzero entries. What is the operation bound of the sparse product? What fraction of the number of multiplications of the classical product is it?

Answer

The bound of the sparse product is $cols_B \cdot t_A + rows_A \cdot t_B = 1000 \cdot 3000 + 1000 \cdot 3000 = 6 \cdot 10^6$. The classical product makes $1000^3 = 10^9$ multiplications. The sparse method does about 167 times less work.

5.15 Further reading

- E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C* (2nd ed.), Sections 2.4–2.5 and 4.4, 4.7: the source of the `padd`, `transpose`, `fast_transpose` and `mmult` code of these notes; it gives the linked polynomial and the linked sparse matrix built with circular lists in

detail.

- Y. Saad, *Iterative Methods for Sparse Linear Systems* (2nd ed.), Section 3.4: describes CSR, CSC and other sparse storage formats together with their use in scientific computing.
- T. H. Cormen et al., *Introduction to Algorithms* (3rd ed.), Section 8.2: gives counting sort, on which the fast transpose rests, with its stability and linear-time analysis.

5.16 References

1. E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2nd ed., Silicon Press, 2008 (Sections 2.4–2.5: polynomials and sparse matrices; 4.4 and 4.7: linked polynomials and sparse matrices).
2. Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed., SIAM, 2003 (Section 3.4: CSR/CSC formats).
3. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 3rd ed., MIT Press, 2009 (Section 8.2: counting sort).

PART II · CHAPTER 6

Stacks

A stack is the simplest linear structure: items are inserted and deleted at one end only, and the item added last is the first to leave. At first glance this restriction looks like a weakness. Yet whenever a problem obeys the rule "the task opened last is closed first," the stack is exactly the right tool: function calls, bracket matching, the undo command, and backtracking all follow that rule. These notes first define the stack as an abstract data type. They then implement the same interface in three ways: with a fixed array and a `top` index, with an array that grows on demand, and with a linked list. For each implementation, diagrams show what happens in memory and trace tables follow every line of code. The last sections cover the computer's own stack, the call stack, then keeping several stacks in a single memory array, and finally which output orders a stack can produce.

LEARNING OUTCOMES

Define LIFO

Define a stack as a list $S = \{a_0, \ldots, a_{n-1}\}$, identify its bottom and top items, and find the state of a stack after a sequence of operations.

Implement it with an array

Write push and pop routines around a `top` index; check for overflow and underflow; explain the difference between `++*top` and `*top--`.

Build a linked stack

Allocate nodes with `malloc` and link them, release popped nodes with `free`, and empty the stack without leaking memory.

Understand the call stack

Explain how function calls behave like a stack, how each call pushes a frame, and why the call stack overflows.

Make a design decision

Choose among a fixed array, a growable array, and a linked list by time, memory, and risk of bugs; justify the amortized cost.

Manage multiple stacks
Share one memory array among n stacks with the arrays `top[i]` and `boundary[i]`, and recognize the limits of this scheme.

6.1 The stack: last in, first out

Arrays and linked lists allow access to items anywhere. In an array we can write to any index; in a list we can insert a node after any node. A stack deliberately restricts this freedom. Items enter at one end only and leave from the same end. This section defines the stack and shows how it grows and shrinks over a sequence of operations.

6.1.1 Definition

Everyday life offers many stack-like arrangements. From a pile of plates in the kitchen we take the top plate first. A new document on a desk goes on top of the pile and is read first. Figure 6.1 gives a more instructive example: a dead-end street one car wide, closed by a wall at one end. Cars have entered the street in the order A, B, C, D. The first car that can leave is D, the one nearest the entrance. For A to leave, D must leave first, then C, then B. Because one end is closed, cars must leave in the reverse of the order in which they entered.

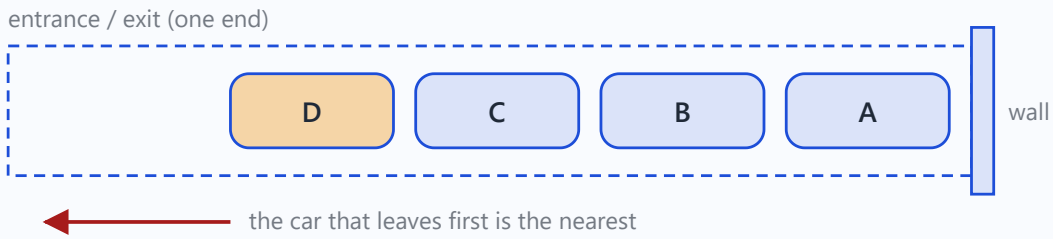


Figure 6.1: A one-car-wide dead-end street. Cars enter and leave at the same end. D, the last to enter, is nearest the entrance and must leave first; A, the first to enter, leaves last.

Definition 6.1

A **stack** is an ordered list in which insertion and deletion take place at one end only. In a stack $S = \{a_0, a_1, \ldots, a_{n-1}\}$, the **bottom** item is a_0 and the **top** item is a_{n-1}. Item a_i sits on top of a_{i-1} ($0 < i < n$). A new item is always placed above a_{n-1}, and the item removed is always the current top. The first item out is therefore the last one in. This principle is called **last in, first out** (**LIFO**).

The key word in the definition is "only." The items inside a stack are ordered, but from outside that order can be touched at a single point. Adding an item is called **push** and removing the top item is called **pop**. These are the standard names, and the notes use them throughout. Some textbooks,

including Horowitz and Sahni, call the same operations *add* and *delete*.

Be careful not to confuse the word "stack" with two other things that share its name or spelling. The *call stack* (Section 6.7) is the stack the system uses to run function calls. The *heap* is the region of memory from which `malloc` allocates blocks, and it is also the name of the priority-queue data structure studied in a later chapter. None of these is related to the others except by name. In this chapter, "stack" means the abstract LIFO structure unless the call stack is mentioned explicitly.

6.1.2 How a stack grows and shrinks

Push A, B, C, D, and E, in that order, onto an empty stack. Keep the stack in an array and store the index of the top item in a variable named `top`. In an empty stack `top` is −1. Each push increments `top` by one and writes the new item at that index. Figure 6.2 shows three moments in this process, followed by the first deletion.

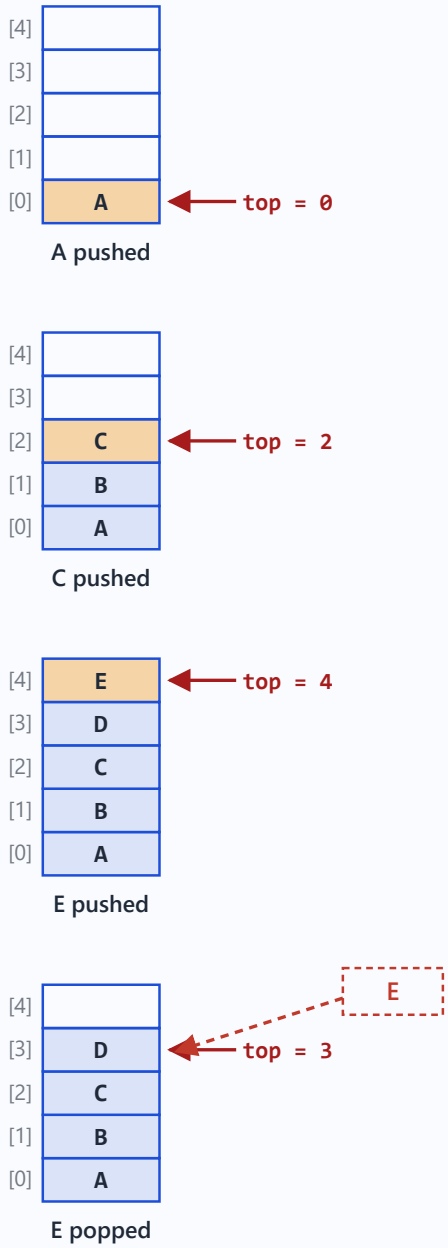


Figure 6.2: Three moments while A, B, C, D, E are pushed onto an empty stack, and the first deletion. After A is pushed, `top` = 0; after C, `top` = 2; after E, `top` = 4. Deletion happens only at the top: the first item out is E, the most recently added, and `top` drops back to 3; the top item is now D.

After every push the top item changes: after A is pushed the top is A, after B is pushed it is B, and so on. When E has been pushed, the stack holds five items and `top` is 4. The number of items is always `top + 1`. The first deletion that follows returns E, because E is on top. Had all items been popped one by one, the output order would have been E, D, C, B, A, which is exactly the reverse of the input order.

An item in the middle of the stack, say B, cannot be reached directly. To get B, the items above it, C, D, and E, must be popped first. This restriction is not a defect but a contract. A program that uses a stack is dealing with a problem in which the rule "the task opened last is closed first" holds. Because the structure permits nothing but that rule, the program stays simple and mistakes become harder to make. Wherever a stack is used, this rule is at work.

Example 6.1

The following operations are applied to an empty stack: push(3), push(8), pop, push(5), push(1), pop, pop, push(9). After each step, the stack (bottom to top) and the item popped are:

1. push(3): [3]
2. push(8): [3, 8]
3. pop: 8 comes out; [3]
4. push(5): [3, 5]
5. push(1): [3, 5, 1]
6. pop: 1 comes out; [3, 5]
7. pop: 5 comes out; [3]
8. push(9): [3, 9]

The items popped are 8, 1, 5, in that order. At the end the stack holds two items; the bottom item is 3, the top item is 9, and in an array implementation `top` = 1. The item 3 is never popped, because it stays at the very bottom throughout the sequence.

6.2 The stack abstract data type

Asking first what a data structure does and only then how it does it makes a program easier to change. An **abstract data type** (**ADT**) defines a set of values and the operations on those values, independently of how they are implemented. The stack ADT says which operations a stack offers and under what conditions they fail. It does not say whether the stack is built from an array or a list.

6.2.1 Operations and preconditions

Table 6.1 lists the six operations of the stack ADT. The words `push`, `pop`, and `peek` beside the operations are the common names for the same operations.

Operation	Meaning	Precondition / error
`stack_create(max)`	Creates an empty stack that holds at most `max` items	—
`stack_full(S)`	Returns true if the stack holds `max` items	—

Operation	Meaning	Precondition / error
`push(S, x)`	Puts `x` on top	Overflow if the stack is full
`stack_empty(S)`	Returns true if the stack holds no items	—
`pop(S)`	Removes and returns the top item	Underflow if the stack is empty
`stack_top(S)` · peek	Reads the top item without removing it	Error if the stack is empty

Table 6.1: The stack abstract data type.

The two error conditions in the table have names. Trying to add an item to a full stack is called **overflow**; trying to take an item from an empty stack is called **underflow**. Overflow rests on the assumption that a stack has a capacity. The abstract stack on paper is unbounded, but every real memory is finite. That is why the ADT gives `stack_create` a `max` value and defines the query `stack_full`.

Underflow, by contrast, has nothing to do with capacity. An empty stack has no top item, so asking for one is meaningless. The precondition of `pop` and `stack_top` is "the stack is not empty." Who checks this precondition is a design decision. The caller can ask `stack_empty` before every `pop`. The operation itself can also check and return an error code. The second way is safer, because forgetting the check becomes impossible.

The real value of an ADT is that it separates the code that uses a structure from the code that implements it. The same six operations can be implemented with a fixed array, with an array that grows as needed, or with a linked list. A program that calls only the ADT operations does not change when the implementation changes. These notes build all three implementations in turn and compare the costs of the same operations.

Common mistake

Confusing `pop` with `peek`. `pop` removes the top item and the stack shrinks by one. `peek` (`stack_top`) only reads the value of the top item and the stack is unchanged. If a loop that merely inspects the top value calls `pop` instead of `peek`, an item disappears at every look.

6.2.2 Where stacks are used

The stack is one of the most widely used structures in computer science. What its applications share is that tasks are opened and closed in a nested fashion: the inner task finishes before the outer one. In each of the following examples, the top of the stack holds "the innermost task that is open right now."

Call stack
: Each function call pushes a frame, containing the return address and the local variables, onto the stack. When the function returns, the frame is popped. The function called last finishes first.

Expression evaluation

Compilers first convert an infix expression such as `a + b * c` to postfix notation. The postfix form of this expression is `a b c * +`. The compiler then evaluates that notation with a stack. In a postfix expression, each number is pushed, and each operator takes the top two values and pushes the result.

Backtracking

A program searching for a path through a maze puts each fork on a stack. On reaching a dead end, it returns to the fork on top of the stack, the most recently passed one, and tries another path from there.

Bracket matching

Every closing bracket in an expression must match the most recently opened bracket that is still unclosed. The phrase "most recently opened" points straight to a stack.

Undo

A text editor puts every action it performs on a stack. The undo command (Ctrl+Z) reverses the action on top of the stack, the most recent one. A second press reverses the action before it.

Depth-first search

Depth-first search (DFS) in a graph keeps the nodes to be visited on a stack. This way the search goes as deep as possible along the most recently discovered path.

The interactive stack tool in the slides demonstrates this rule with a single button. When it pushes four times and then pops four times, the output order is the reverse of the input order. The fifth pop acts on an empty stack and the tool reports it as underflow.

6.2.3 Checking balanced brackets

One application shows in a few lines of code why the stack is the natural choice. Suppose we ask whether the brackets `(`, `[`, and `{` in a string are balanced. In a balanced string every closing bracket has a matching opener, the types agree, and no bracket is left open at the end. `{a[i] = (b + c) * d;}` is balanced. `([)]` is not: the counts are fine, but when `)` arrives, the most recently opened bracket is `[`.

The function below uses a `Stack` type that keeps an array and its top index in one structure. `push` and `pop` return success as a `bool`; `pop` writes the removed value to the location pointed to by its second argument.

```
/* balance check for (, [ and {: each closer must match the latest opener */
bool balanced(const char *s) {
    Stack st;  int opener;
    stack_init(&st);
    for (int i = 0; s[i] != '\0'; i++) {
        char c = s[i];
        if (c == '(' || c == '[' || c == '{') {
            if (!push(&st, c)) return false;       /* nested too deep */
        } else if (c == ')' || c == ']' || c == '}') {
            if (!pop(&st, &opener)) return false;  /* closer with no opener */
            if ((c == ')' && opener != '(') || (c == ']' && opener != '[') ||
                (c == '}' && opener != '{'))
                return false;                      /* types do not match */
        }
    }
    return stack_empty(&st);                       /* anything left open? */
}
```

The function scans the string once from left to right. On seeing an opening bracket, it pushes it. On seeing a closing bracket, it pops the opener from the top of the stack. If the stack is empty at that moment, the closing bracket has no opener and the result is false. If the type of the popped opener does not agree with that of the closer, the result is again false. Characters that are not brackets are skipped. If items remain on the stack when the scan ends, some brackets were never closed, so the function finally returns whether the stack is empty.

The function was tried on these inputs: it returned 1 (true) for `{a[i] = (b + c) * d;}` and `x = f(a[2], {1, 2});`, and 0 (false) for `([)]`, `((`, `())`, and `]`. The empty string is balanced and returns 1. Each character causes at most one push and one pop. Since push and pop take constant time, a string of n characters takes $\Theta(n)$ time. In the worst case every character is an opening bracket and the stack holds n items, so the extra memory is $O(n)$.

6.3 Array-based stack

The shortest way to implement a stack is a one-dimensional array. Cell 0 of the array is the bottom of the stack. Items are laid out upward through cells 1, 2, 3, and so on. The index of the top item is kept in a separate integer variable, `top`. This section first explains what `top` means and its two boundary cases, then goes through the C code of Horowitz and Sahni line by line.

6.3.1 The `top` index, overflow, and underflow

In an array stack, all the information lies in two things: the contents of the array and the value of `top`. The variable `top` is the index of the item on top of the stack. If the stack holds k items, they occupy cells `stack[0]` through `stack[k-1]` and `top` = $k - 1$. When the stack is empty there are no

items; in that case we choose `top` = −1. The value −1 is not a valid index, but at the first push, incrementing `top` by one gives exactly 0. This choice simplifies the code.

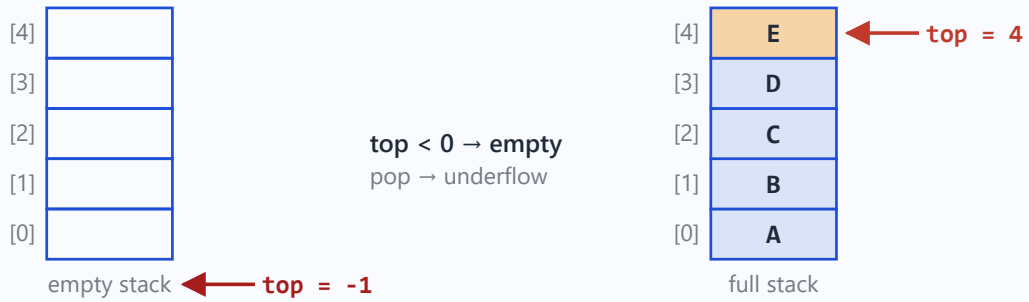


Figure 6.3: The two boundary states of an array stack with five cells. On the left, the empty stack: `top` = −1, below even cell 0; `pop` causes underflow in this state. On the right, the full stack: five items A through E, and `top` = 4 = `MAX_STACK_SIZE - 1`; one more push would cause overflow.

Figure 6.3 shows the two boundary states side by side. Whether the stack is empty is determined by the test `top < 0`. If the array has `MAX_STACK_SIZE` cells, the last valid index is `MAX_STACK_SIZE - 1`. When `top` reaches this value the stack is full, and the test `top ≥ MAX_STACK_SIZE - 1` is true. The number of items is always `top + 1`. Confusing `top` with the item count is the best-known source of off-by-one errors. For example, a program that writes the full test as `top ≥ MAX_STACK_SIZE` writes one cell past the end of the array.

C does not check array bounds at run time. The expression `stack[MAX_STACK_SIZE]` compiles and runs, but it touches the memory cell right after the array. If another variable lives there, its value is corrupted. The program may not crash immediately, and the error shows up much later in an unrelated place. For this reason the overflow and underflow checks are not an optional decoration of stack code. A push without a check overwrites neighboring memory.

6.3.2 C definitions

The first implementation in the textbook by Horowitz and Sahni defines the stack as a global array:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_STACK_SIZE 101

typedef struct {
    int key;
    /* other fields */
} element;

element stack[MAX_STACK_SIZE];

void stack_full(void) {
    printf("Stack is full!!!\n"); exit(-1);
}
void stack_empty(void) {
    printf("Stack is empty!!!\n"); exit(-2);
}
```

The line `#define MAX_STACK_SIZE 101` fixes the capacity of the stack. Before compilation, the preprocessor replaces every occurrence of `MAX_STACK_SIZE` in the code with 101. To change the capacity, it is enough to edit this one line.

The type `element` defines the record stored in the stack. For now it has a single field, `key`. The comment is a reminder that in a real application other fields would be added to this record. A maze program, for instance, would put the row, the column, and the direction on the stack together. Defining the item as a `struct` rather than an `int` makes the stack code independent of what an item contains. When fields are added to the record, the stack routines do not change, because they merely copy `element` values.

The line `element stack[MAX_STACK_SIZE];` is the stack itself. Because this array is defined outside every function, it is a **global variable**. Every function in the program can access it directly. Global arrays are initialized to zero and live for the whole run of the program. The variable `top`, however, is not global: it is defined as a local variable in the calling function (usually `main`) and passed to the stack routines by address.

The last two functions are error routines. `stack_full` prints a message and ends the program with `exit(-1)`, and `stack_empty` does so with `exit(-2)`. `exit` is declared in `<stdlib.h>` and ends the whole program no matter which function it is called from. The number passed is the exit code of the program and is handed to the operating system. On Unix-like systems the shell sees −1 as 255; on Windows this code is reported as 4294967295. The names of these two functions are the same as those of the queries `stack_full` and `stack_empty` in the ADT, but their jobs differ: they do not query, they report an error. The name clash is one of the flaws of this first implementation; in a better design these names go to the real queries.

Textbooks give these definitions in a single file, with a global array and a global `top`. In the exams and assignments of this course, however, global variables are forbidden. The reason is simple. Any function of the program can change a global variable. When a bug appears, finding who corrupted the variable means reading the whole program. Moreover, a single global array is useless when the same program needs two stacks. In this chapter the global array is used only to get to know the textbook code; in assignments, whatever the structure is, a stack, a queue, or something else, it is defined locally and passed by parameter.

6.3.3 `push` and `pop`

The insertion and deletion routines are three lines each:

```
void push(int *top, element item) {
    if (*top >= MAX_STACK_SIZE - 1)  /* any room? */
        stack_full();          /* warn and exit */
    stack[++*top] = item;      /* bump the index first */
}                              /* then store the item */

element pop(int *top) {
    if (*top < 0)
        stack_empty();         /* warn and exit if empty */
    return stack[(*top)--];    /* return, then decrement */
}
```

`push` takes two parameters. `int *top` is the address of the caller's `top` variable. `element item` is a copy of the item to be added. The first line tests whether there is room. `*top` is the integer that the pointer `top` points to, that is, the caller's top index. If this value has reached `MAX_STACK_SIZE - 1`, the stack is full and `stack_full()` ends the program. If there is room, the last line runs: `stack[++*top] = item;`. This line does two things. First `*top` is incremented, then the item is written at the new index. In an empty stack `*top` goes from −1 to 0 and the item is written to `stack[0]`.

`pop` takes only the address of `top`. If the stack is empty, that is, `*top < 0`, `stack_empty()` ends the program. Otherwise the line `return stack[(*top)--];` runs. This line also does two things, but in the opposite order. First the item at the old index is read, then `*top` is decremented. The item read is the return value of the function. For example, when `*top` is 2, `stack[2]` is returned and `*top` becomes 1.

The popped item is not erased from the array. The old value stays in cell `stack[2]`. But it is no longer part of the stack, because `top` determines the contents of the stack: the valid items are the cells from 0 up to `top`. The next push overwrites this cell. That deletion means pulling the boundary down rather than clearing a cell is what makes the array stack fast.

Why is `top` passed by address? In C, all arguments are passed by **call by value**. A call such as `push(top, item)` would hand the function a copy of the value of `top`. The function would increment this copy, but the caller's `top` would not change. The next push would write to the same cell. A function that wants to change the caller's variable is given the address of that variable: `push(&top, item)`. Through `*top` the function reaches the caller's actual variable.

A writing rule matters here. A parameter that appears as `*top` in the function's parameter list must be used as `*top` everywhere inside the function. If only `top` is written inside, the pointer itself, an address, is being used. A test such as `if (top >= MAX_STACK_SIZE - 1)` compares an address with an integer. At best the compiler gives a warning; the program makes a meaningless comparison. This mistake is graded severely in exams, because not knowing the difference between a pointer and the value it points to means building every structure in this course wrongly.

Both routines cost *O(1)*. Each performs one comparison, one index change, and one array access. The number of these operations does not depend on the number of items in the stack. The extra memory is also constant: only the parameters are used.

6.3.4 The precedence trap: `*` with `++` and `--`

The expressions `stack[++*top]` and `stack[(*top)--]` are short but demand attention. The first has no parentheses; the second does. The reason for this difference is operator precedence in C. Table 6.2 compares four spellings.

Expression	How C parses it	Result
`++*top`	`++(*top)` : increment the int it points to, use the new value	Correct (push)
`(*top)--`	use the value of the int it points to, then decrement it	Correct (pop)
`*top--`	`*(top--)` : decrement the pointer, read the value at the old address	Wrong: `*top` is unchanged
`*top++`	`*(top++)` : advance the pointer, read the value at the old address	Wrong

Table 6.2: What four spellings mean when `top` is an `int *`.

The rule is this. The postfix operators `++` and `--` have higher precedence than the unary `*`. So in the expression `*top--`, the `--` binds to `top` first: what gets decremented is the pointer itself. The prefix `++` and the unary `*`, on the other hand, are at the same precedence level and associate from right to left. In the expression `++*top`, the `*` on the right is applied first, and `++` binds to its result, the integer being pointed to. That is why push needs no parentheses and pop does.

What happens if we write `*top--` instead of `(*top)--`? The expression reads as `*(top--)`. `top--` yields the old value of the pointer and moves the pointer back by one `int`. The `*` reads at this old address. The value read is the caller's top index, so the array is read from the right cell. But what gets decremented is the local `top` pointer. When the function ends, this local copy vanishes. The caller's `top` has not changed at all. A bug of this kind is very insidious for exactly this reason: the first `pop` returns the right value.

Example 6.2

The stack contains only A (`top` = 0). The last line of `pop` has mistakenly been written as `return stack[*top--];`. Two calls are made in a row.

1. First call: `*(top--)` reads the caller's 0. `stack[0]` = A is returned. What is decremented is the local pointer; the caller's `top` stays 0.
2. Second call: `*top < 0` is false, so underflow is not caught. Again `stack[0]` = A is returned, and `top` is again 0.

This behavior was verified by compiling and running the code: both lines printed "pop: A top=0," and the compiler gave no warning even with `-Wall -Wextra`. A program that empties the stack with a loop of the form `while (!empty) pop` goes into an infinite loop. The `*top++` mistake is worse in push: `stack[*top++] = item` writes to `stack[-1]` in an empty stack, that is, outside the array.

The most robust way to avoid this trap is to put the increment or decrement through the pointer on a separate line: first `(*top)++;`, then `stack[*top] = item;`. The code gets one line longer, but everyone who reads it understands at once what happens. The short form in the textbook is correct too, as long as one knows why the parentheses are there.

6.3.5 Code trace

The behavior of the routines is easier to see with a small capacity. In the trace `MAX_STACK_SIZE` = 4, so the full test is `*top ≥ 3`. In `main`, with `top` = −1, A, B, and C are pushed in turn, one item is popped, and then D, E, and F are pushed. Table 6.3 gives the state after each call.

Call	Test	Action	`top`	`stack[0..top]`
start	—	—	−1	(empty)
`push(&top, A)`	−1 ≥ 3? No	`++*top`, `stack[0] = A`	0	A
`push(&top, B)`	0 ≥ 3? No	`stack[1] = B`	1	A B
`push(&top, C)`	1 ≥ 3? No	`stack[2] = C`	2	A B C
`pop(&top)`	2 < 0? No	returns `stack[2]` = C, `top` decrements	1	A B
`push(&top, D)`	1 ≥ 3? No	`stack[2] = D` (over C)	2	A B D
`push(&top, E)`	2 ≥ 3? No	`stack[3] = E`	3	A B D E
`push(&top, F)`	3 ≥ 3? **Yes**	`stack_full()`: "Stack is full!!!", `exit(-1)`	3	A B D E

Table 6.3: Trace of the array stack with `MAX_STACK_SIZE` = 4.

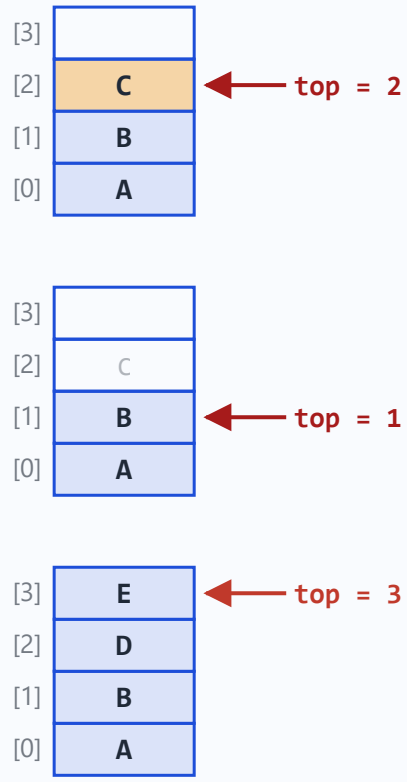


Figure 6.4: Three moments of the trace. At the top, C has been pushed: `top` = 2. In the middle, C has been popped: `top` has dropped to 1; the value of C stays in the cell but is no longer part of the stack. At the bottom, D and E have been pushed: `top` = 3 and the array is full; when F is to be pushed, `stack_full()` is called.

The fifth row of the table shows how the array performs deletion. `pop` returns C and lowers `top` to 1. C still sits in cell 2. In the sixth row, when D is pushed, D is written to the same cell and no trace of C remains. In the last row there is no room. `stack_full` prints its message and ends the whole program with `exit(-1)`. This behavior was verified by compiling and running the code: the program printed "push F ... Stack is full!!!" and Windows reported the exit code as 4294967295, that is, −1.

Ending the whole program at the moment of overflow is a crude reaction. A stack is a small library component; it cannot know what the rest of the program will do. Perhaps the caller was going to build a larger stack when this one filled up. Perhaps it would show the user a message and cancel the operation. `exit` takes these decisions away. Cleanup work such as closing open files properly or releasing allocated memory is also left undone when the program ends. A better design reports the error to the caller and leaves the decision to it.

6.4 Designing the stack as an object

The first implementation in the textbook has three weaknesses. The stack is a global array, so the program can have only one stack. The variable `top` lives apart from the array; the caller has to define it separately and pass it separately in every call. On an error, the program ends with `exit`. This section builds a design that solves all three problems together: the array and the top index live in one structure, the routines take the stack as a parameter, and errors are reported by a return value.

6.4.1 A `struct` instead of a global array

```
#include <stdbool.h>
#define CAP 100

typedef struct {
    int data[CAP];
    int top;               /* -1 when empty */
} Stack;

void stack_init(Stack *s) { s->top = -1; }
bool stack_empty(const Stack *s) { return s->top < 0; }
bool stack_full(const Stack *s)  { return s->top == CAP - 1; }
int  stack_size(const Stack *s)  { return s->top + 1; }
bool stack_top(const Stack *s, int *x) {
    if (stack_empty(s)) return false;
    *x = s->data[s->top];  return true;
}
```

The `Stack` structure carries two fields: the array `data` of `CAP` items and `top`, which holds the top index. A stack is now a single variable. The definition `Stack a, b;` gives two independent stacks, each with its own array and its own top index. While one fills up, the other is unaffected. There are no global variables.

Each routine receives the stack through the parameter `Stack *s`, that is, by address. There are two reasons for this. First, `push` and `pop` modify the stack; if a copy passed by value were modified, the caller's stack would not change. Second, `Stack` is a large structure: with `CAP` = 100 it takes about 404 bytes. Passed by value, these 404 bytes would be copied at every call. An address is 8 bytes. The fields of the structure are accessed through the address with the arrow operator (`->`): `s->top` is shorthand for `(*s).top`.

The queries that do not modify the stack take `const Stack *s`. The `const` qualifier is a promise that the function will not modify the structure it points to. If an assignment such as `s->top = 0;` is written inside the function, the compiler reports an error. This promise informs the reader and also catches accidental modifications at compile time.

Each routine is a single line. `stack_init` creates an empty stack by setting `top` to −1. The fields of a local `Stack` variable hold indeterminate values at first; so every stack must be initialized once before use. `stack_empty` and `stack_full` are now real queries and return `bool`. `stack_size` gives the item count, `top + 1`. `stack_top` is the peek operation of the ADT. If the stack is empty it returns `false` and leaves `*x` untouched. Otherwise it writes the top item to `*x` and returns `true`. The value is delivered through an output parameter because the return value is reserved for success information. The type `bool` and the names `true` and `false` come from the header `<stdbool.h>`.

Push and pop are written in the same style:

```
bool push(Stack *s, int x) {
  if (s->top == CAP - 1) return false; /* overflow */
  s->data[++s->top] = x;
  return true;
}
bool pop(Stack *s, int *x) {
  if (s->top < 0) return false;   /* underflow */
  *x = s->data[s->top--];
  return true;
}
```

`push` first tests for overflow. If `top == CAP - 1` there is no room and the function returns `false`. The program goes on; the caller, having learned of the overflow, decides for itself what to do. If there is room, the line `s->data[++s->top] = x;` first increments `top`, then writes the value at the new index. There is no precedence trap here: the `->` operator is applied before `++`, so `++s->top` means `++(s->top)`.

`pop` first tests for underflow. If the stack is empty it returns `false` and leaves `*x` untouched. Otherwise the line `*x = s->data[s->top--];` copies the top item to `*x` and then decrements `top` by one. The expression `s->top--` likewise reads as `(s->top)--`; what is decremented is a field of the structure.

Because this version reports errors through the return value, it is very convenient to use. For instance, a loop that empties the stack and prints each item popped is a single line: `while (pop(&st, &x)) printf("%d ", x);`. The loop stops when `pop` returns `false` for the first time, that is, when the stack has become empty. All operations are again *O(1)*.

Common mistake

Ignoring the result of a routine that returns `bool`. A program that writes `push(&st, x);` and never looks at the return value silently loses the item when the stack fills up. A program that writes `pop(&st, &x);` without checking the result keeps using the old or indeterminate value of `x` when the stack is empty. An interface that reports errors is useful only if the report is read.

6.4.2 Code trace: `push` and `pop` returning `bool`

In the trace `CAP` = 3, so the full test is `top == 2`. An empty stack is created with `stack_init(&st)`. Then 5, 8, 2, and 9 are pushed in turn, and `pop` is called four times. Table 6.4 gives each call and its return value.

Call	Test	Action	Returns	`top`	`data[0..top]`
`stack_init(&st)`	—	`top = -1`	—	−1	(empty)
`push(&st, 5)`	−1 == 2? No	`data[0] = 5`	true	0	5
`push(&st, 8)`	0 == 2? No	`data[1] = 8`	true	1	5 8
`push(&st, 2)`	1 == 2? No	`data[2] = 2`	true	2	5 8 2
`push(&st, 9)`	2 == 2? **Yes**	overflow, nothing is written	false	2	5 8 2
`pop(&st, &x)`	2 < 0? No	`x = data[2]`	true, `x` = 2	1	5 8
`pop(&st, &x)`	1 < 0? No	`x = data[1]`	true, `x` = 8	0	5
`pop(&st, &x)`	0 < 0? No	`x = data[0]`	true, `x` = 5	−1	(empty)
`pop(&st, &x)`	−1 < 0? **Yes**	underflow, `x` untouched	false	−1	(empty)

Table 6.4: Trace of `push` / `pop` with `CAP` = 3.

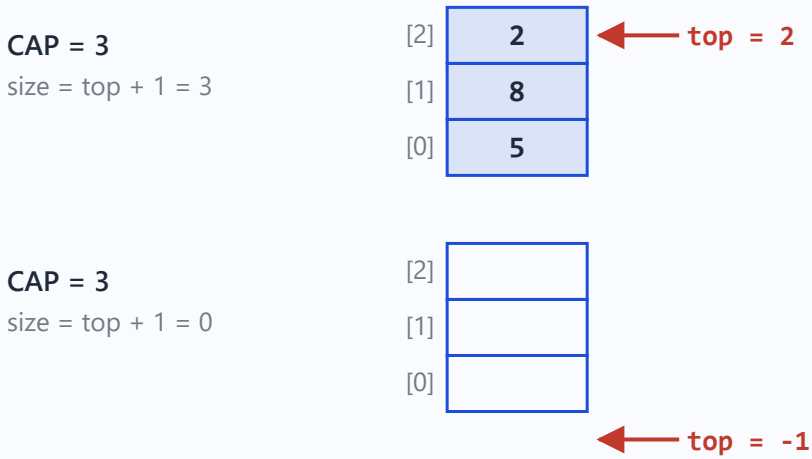


Figure 6.5: Two moments of the trace. At the top, after three pushes the stack is full: `top` = 2 = `CAP - 1`, the size is 3; the fourth push returns `false`. At the bottom, after four pops the stack is empty: `top` = −1 and the last pop has returned `false`.

The trace was verified with a test program. The program, which printed the return value of each push, produced the output "push 5:1 8:1 2:1 9:0"; 9 could not be added. The pops returned 2, 8, 5 in that order, and the fourth pop was `false`. The intermediate queries also worked as expected: after three pushes, `stack_size` gave 3, `stack_full` was true, and `stack_top` gave 2.

The difference between the two implementations shows at the overflow line. The global-array version would end the program at this point. The `bool` version returns `false` and the program continues where it left off. In the example, the caller learns that 9 could not be added. It may print a

warning to the user or place 9 somewhere else. This way a single routine does not decide the fate of the whole program.

6.4.3 Interface and implementation

The `struct` version still has a problem. The fields of the `Stack` structure are open to all the code that uses it. A programmer can write `st.top = 5;` and corrupt the stack, or reach into `st.data[0]` directly and break the stack discipline. A bigger problem is this: when the implementation of the stack changes, say from an array to a linked list, all the code that touches the fields directly breaks.

In C, this problem is solved by splitting the code into a **header file** and a source file. The header file carries the **interface**, that is, the types and function declarations exposed to the outside. The source file carries the **implementation**, that is, the bodies of these functions and the inside of the structure.

```
/* stack.h: all the client sees */
#ifndef STACK_H
#define STACK_H
#include <stdbool.h>
typedef struct Stack Stack;  /* opaque */
Stack *stack_create(int capacity);
void stack_destroy(Stack *s);
bool push(Stack *s, int x);
bool pop(Stack *s, int *x);
bool stack_empty(const Stack *s);
#endif
```

The first and last lines of the header file are an **include guard**. In a large program, the same header can be included in a source file twice through indirect paths. The test `#ifndef STACK_H` processes the contents of the file if the name `STACK_H` is not yet defined, and defines this name right after. On the second inclusion the name is already defined, so the contents are skipped. This prevents the error of defining the same type twice.

The really important line is `typedef struct Stack Stack;`. This line says that a structure named `struct Stack` exists but does not say what its fields are. A type whose name alone is declared in this way is called an **incomplete type**. A type whose fields are hidden from the client is also called an **opaque type**. Client code cannot define a variable of an incomplete type, because the compiler does not know how many bytes it takes. Nor can it access its fields. The only thing it can do is hold a pointer to this type and pass the pointer to the functions in the header. That is why a stack is created with `stack_create` and destroyed with `stack_destroy`.

```c
/* stack.c: implementation */
struct Stack { int *data; int top, cap; };

Stack *stack_create(int cap) {
  Stack *s = malloc(sizeof *s);
  if (s == NULL) return NULL;
  s->data = malloc(cap * sizeof *s->data);
  if (s->data == NULL) {
    free(s);  return NULL;
  }
  s->top = -1;  s->cap = cap;
  return s;
}
void stack_destroy(Stack *s) {
  if (s) { free(s->data); free(s); }
}
```

The source file holds the full definition of the structure. In this version the array is not of fixed size inside the structure. `data` is a pointer, and the array itself is allocated from heap memory during `stack_create`, with the requested capacity. This way the capacity of each stack can be chosen separately.

`stack_create` performs two allocations. First `malloc(sizeof *s)` is called for the structure itself. `sizeof *s` is the size of the type that `s` points to, that is, `struct Stack`. This spelling stays correct even if the type name changes. If the allocation fails, `NULL` is returned. Then `malloc(cap * sizeof *s->data)` is called for the array of `cap` items. If this second allocation fails, there is an important detail: the structure block allocated first is returned with `free(s)`, and only then is `NULL` returned. If this line were forgotten, the structure's block would remain as a piece of memory that no variable can point to; that is, a **memory leak** would occur. If both allocations succeed, `top` is set to −1, `cap` to the requested capacity, and the address of the structure is returned.

`stack_destroy` follows the reverse of the allocation order. First the array and then the structure is released. The order matters: if the structure were released first, the field `s->data` would be read from memory that is no longer valid. The test `if (s)` allows the function to be called safely with `NULL` as well. By the C standard, `free(NULL)` does nothing anyway; but to read `s->data`, `s` must be valid.

For brevity, the implementation file in the slides omits the opening `#include` lines and three routines. A complete `stack.c` starts with the lines `#include <stdlib.h>` (for `malloc` and `free`) and `#include "stack.h"` (so that the compiler checks that declarations and definitions agree). The missing routines are the same as `push`, `pop`, and `stack_empty` of the `struct` version; the only difference is that the full test uses the structure's `cap` field instead of the constant `CAP`: `if (s->top == s->cap - 1) return false;`. With these completed, a test program that creates two separate stacks (capacities 2 and 5) was compiled and run. The third push to the stack of capacity 2 returned `false`; after both stacks were destroyed, a counting `malloc`/`free` wrapper counted four allocations and four releases. There is no leak.

The value of this split shows when the implementation changes. Suppose `stack.c` is converted from an array to a linked list. The inside of `struct Stack` changes completely; the fields `data` and `cap` disappear and a node pointer takes their place. But `stack.h` stays the same. Therefore all the files that use the stack keep working without any change, merely being relinked. This is how the ADT's separation of "what" from "how" takes concrete form in C.

6.4.4 Reversing a string with a stack

The simplest application of a stack is reversing a sequence. Items are pushed in order and then popped in order. Because a stack gives back the last item first, the output order is the reverse of the input. The function below reverses a string in place this way; it uses the `push` and `pop` routines of the `struct` version.

```
void reverse(char *s) {
  Stack st;  int c, i;
  stack_init(&st);
  for (i = 0; s[i] != '\0'; i++)
    push(&st, s[i]);    /* assume CAP suffices */
  for (i = 0; pop(&st, &c); i++)
    s[i] = (char)c;
}
```

The function has a local variable `Stack st`. The stack lives in the function's frame and vanishes by itself when the function returns; neither `malloc` nor `free` is needed. The first loop pushes every character up to the `'\0'` at the end of the string. The return value of `push` is not checked here; the assumption in the comment is that the string is shorter than `CAP` = 100 characters. The second loop, as long as `pop` succeeds, writes the popped character to position `i` of the string and increments `i`. When the stack is empty, `pop` returns `false` and the loop ends.

The variable `c` is an `int` because `pop` expects an `int *` as its second argument. In the call `push(&st, s[i])` the `char` value is widened to `int` automatically. When writing back into the string, the cast `(char)c` narrows the value to `char` again. No information is lost, since the characters came from the `char` range in the first place.

Let the input be `"STACK"`. Table 6.5 gives every step of the two loops.

Loop	`i`	Action	Stack (bottom to top)	`s`
start	—	`stack_init`	(empty)	STACK
1	0	push('S')	S	STACK
1	1	push('T')	S T	STACK
1	2	push('A')	S T A	STACK
1	3	push('C')	S T A C	STACK
1	4	push('K')	S T A C K	STACK

Loop	`i`	Action	Stack (bottom to top)	`s`
2	0	pop → 'K', `s[0] = 'K'`	S T A C	KTACK
2	1	pop → 'C', `s[1] = 'C'`	S T A	KCACK
2	2	pop → 'A', `s[2] = 'A'`	S T	KCACK
2	3	pop → 'T', `s[3] = 'T'`	S	KCATK
2	4	pop → 'S', `s[4] = 'S'`	(empty)	KCATS
2	5	pop → `false`, the loop ends	(empty)	KCATS

Table 6.5: Trace of `reverse`, input `"STACK"`.

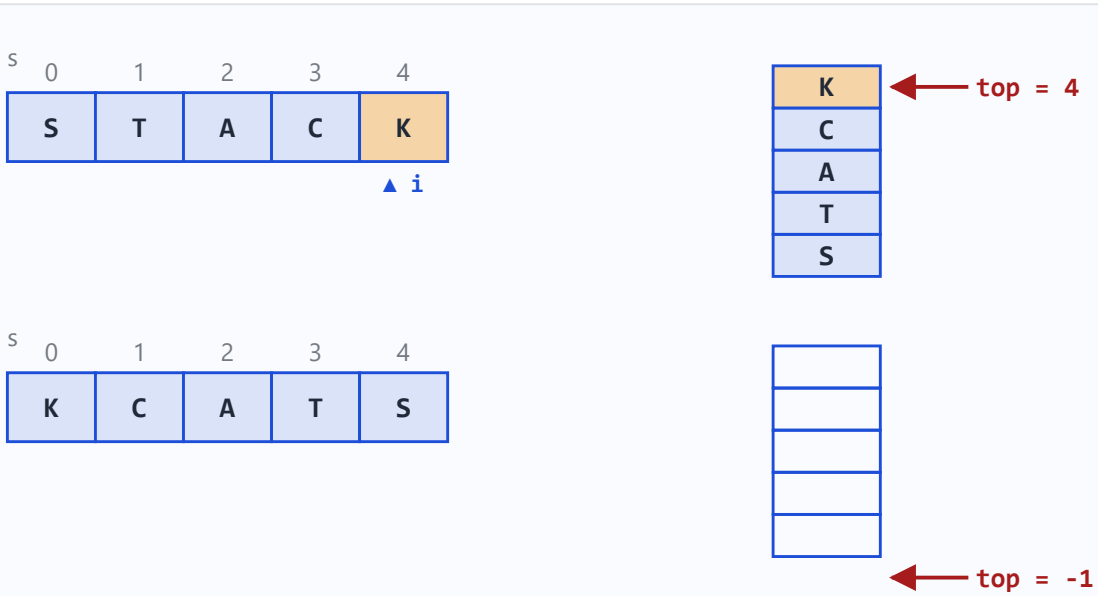


Figure 6.6: Two moments of the reversal. At the top, the first loop has finished: five characters are on the stack with K on top (`top` = 4). At the bottom, the second loop has finished: the stack is empty (`top` = −1) and the string has become KCATS.

The intermediate strings in the table may be surprising at first sight. For two steps `s` reads "KCACK." The reason is that the middle letter of the word "STACK," the A, is written back to the very cell where it already is, so that step changes nothing. Only the other steps actually change the string. The program was run to verify this: it gave KCATS for `"STACK"` and ba for `"ab"`; the empty string stayed empty.

The cost is clear. For a string of n characters, the first loop makes n pushes and the second makes n pops and one failed pop. Since each operation is $O(1)$, the total time is $\Theta(n)$. Because the stack holds all n characters at once, the extra memory is also $\Theta(n)$. The same job can be done with two indices, swapping characters from the two ends, using $O(1)$ extra memory. The value of the stack version is that it shows plainly the stack's ability to reverse order. The same method works for any stream whose order is to be reversed but whose length is not known in advance.

Common mistake

Forgetting the capacity assumption. `reverse` assumes that the string is shorter than `CAP` characters. For a string of 150 characters, the first 100 characters enter the stack, push returns `false` for the remaining 50, and those characters are silently lost. The second loop writes back only 100 characters. The result is a meaningless string made of the reverse of the first 100 characters followed by the remaining 50 characters in their original form. A safe version either checks the result of push and returns an error, or chooses the capacity according to the length of the string.

6.5 Linked stack

The capacity of an array stack is chosen in advance. If the capacity is too small, the stack overflows; if it is too large, unused cells take up space for nothing. A linked list removes this dilemma. A separate node is allocated for each item, and the stack holds exactly as many nodes as it has items. This section builds the stack on a singly linked list.

6.5.1 Why the top is the head of the list

In a linked list each item is a **node**. A node carries two fields: `data`, which holds the value of the item, and `link`, which holds the address of the next node. Because the node type contains a pointer to its own type, it is a self-referential structure:

```
typedef struct node {
    int data;
    struct node *link;
} node;
```

Which end of the stack should correspond to which end of the list? In a singly linked list the two ends are not equal. Inserting at the head and deleting from the head are *O(1)*: only the head pointer and one link change. Insertion at the tail is also *O(1)* if the last node is tracked separately. But to delete from the tail, the link of the second-to-last node has to be set to `NULL`. In a singly linked list one cannot go back from a node to the one before it; the second-to-last node can be found only by walking from the head, in *O(n)* time.

In a stack, insertion and deletion happen at the same end, so that end must be the head of the list. The pointer `top`, which designates the top of the stack, therefore points to the first node of the list. The item added last is at the head of the list and the item added first is at its tail. An empty stack is one whose `top` is `NULL`. In short there are two options: either keep two pointers and track the last node as well, and deletion stays slow anyway, or keep the head with a single pointer and everything is fast. The right choice is the second.

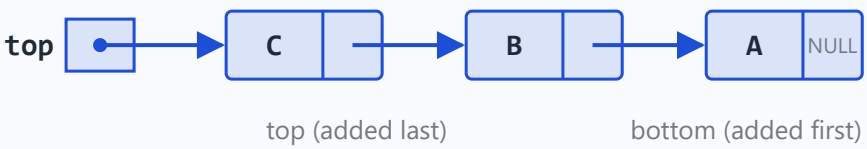


Figure 6.7: A linked stack to which A, B, C have been pushed in that order. `top` points to the head of the list, that is, to C, the item added last; C points to B, and B points to A. The link of A is `NULL`: A is the bottom of the stack.

Figure 6.7 shows a linked stack to which A, B, C have been pushed in that order. Pay attention to the direction of the arrows. They run from top to bottom: each node points to the node that was added before it. A pop moves `top` one arrow forward and removes C from the list. Nothing is shifted, and there is no capacity limit. The stack overflows only when `malloc` cannot find room for a new node.

6.5.2 Linked `push`

```c
bool push(node **top, int x) {
  node *new_node = malloc(sizeof *new_node);
  if (new_node == NULL) return false; /* no memory */
  new_node->data = x;
  new_node->link = *top;  /* point to old top */
  *top = new_node;        /* new top */
  return true;
}
```

The function takes the parameter `node **top`. This is a **pointer to a pointer**: the address of the caller's `top` variable. Push is going to make the caller's `top` point to the new node. If a copy passed by value were changed, the caller's `top` would keep pointing to the old node and the new node would be lost. That is why the call is made as `push(&top, 10)`. Inside the function, `*top` is the caller's `top` variable itself.

The body has four steps:

1. `node *new_node = malloc(sizeof *new_node);` allocates room for one node from heap memory. If the allocation fails, `malloc` returns `NULL` and the function exits with `false`. In that case the stack has not changed at all.
2. `new_node->data = x;` writes the value into the new node. The memory that `malloc` returns is not cleared; the fields hold indeterminate values until this assignment.
3. `new_node->link = *top;` makes the link of the new node point to the old top. If the stack is empty, the old top is `NULL` and the link of the new node becomes `NULL` too.

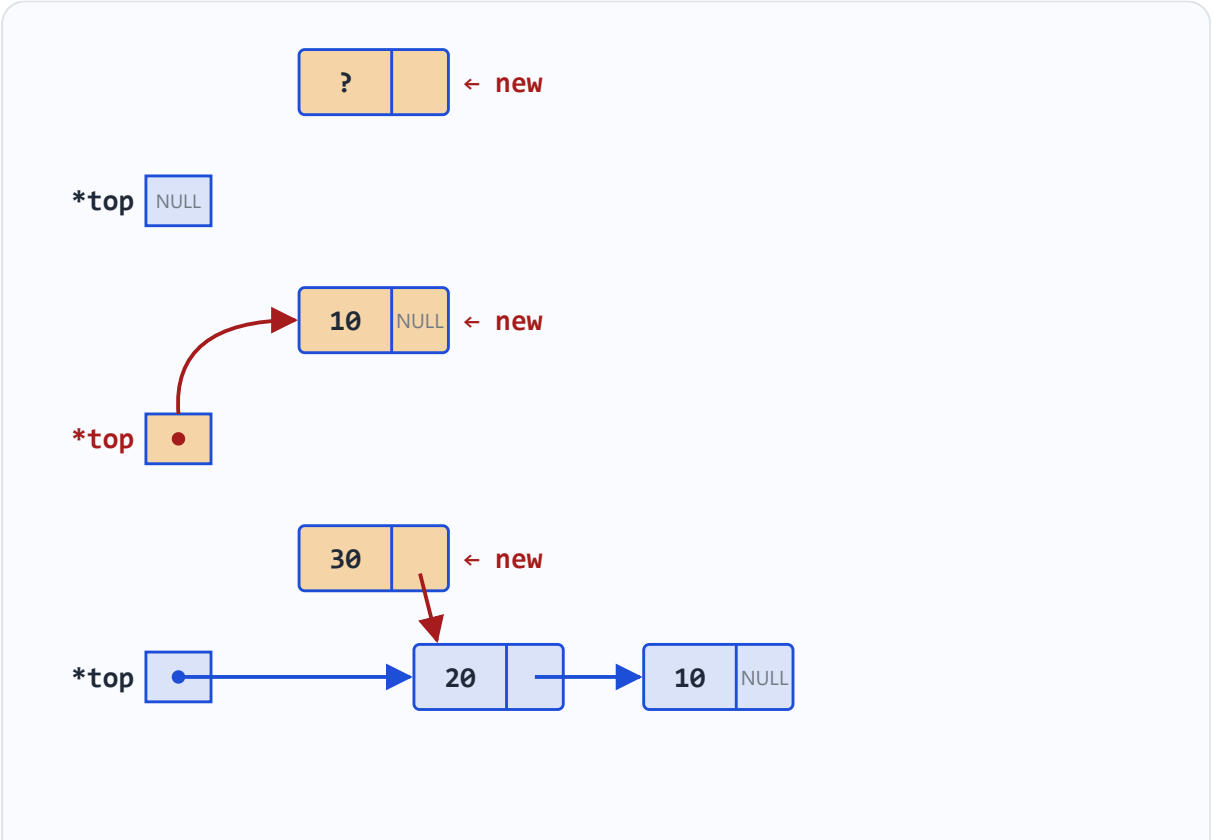
4. `*top = new_node;` makes the caller's `top` point to the new node. The new node is now the top of the stack.

The order of the third and fourth steps is critical. First link, then change the top. Had the order been reversed, the line `*top = new_node;` would have lost the address of the old top. The following `new_node->link = *top;` would then have linked the new node to itself. All the nodes of the old list would have remained as leaked memory that no variable can point to.

Push 10, 20, 30 in turn onto an empty stack. Table 6.6 gives the state after each line. In the table, `node(10)` means the address of the node whose value is 10.

Call	Line	`new_node`	`new_node->link`	`*top`	Stack
start	`node *top = NULL;`	—	—	NULL	(empty)
`push(&top, 10)`	`malloc`	node(?)	?	NULL	(empty)
	`new_node->data = x`	node(10)	?	NULL	(empty)
	`new_node->link = *top`	node(10)	NULL	NULL	(empty)
	`*top = new_node`	node(10)	NULL	node(10)	10
`push(&top, 20)`	`new_node->link = *top`	node(20)	node(10)	node(10)	10
	`*top = new_node`	node(20)	node(10)	node(20)	20 → 10
`push(&top, 30)`	`new_node->link = *top`	node(30)	node(20)	node(20)	20 → 10
	`*top = new_node`	node(30)	node(20)	node(30)	30 → 20 → 10

Table 6.6: Trace of the linked `push` while 10, 20, 30 are pushed onto an empty stack.



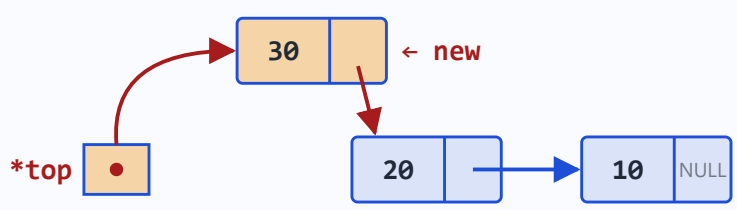


Figure 6.8: Four moments of the linked push. In the first, `malloc` has allocated the first node; its fields are indeterminate (?) and `*top` is still `NULL`. In the second, 10 is the only item of the stack. In the third, the new node holding 30 has been prepared and linked to the old top, 20, but `*top` still points to 20. In the fourth, after `*top = new_node`, the stack is 30 → 20 → 10.

The intermediate state in the third drawing is instructive. The new node is ready and points to 20, but the stack is still 20 → 10. The structure of the stack changes with a single assignment, the line `*top = new_node`. This arrangement is also safe in the failure case: had `malloc` failed, the function would have returned at the first check and the stack would not have been touched at all. Each push does a constant amount of work: one allocation and three assignments. The length of the stack does not affect this amount; push is *O(1)*. The running time of `malloc` itself is also taken to be constant in practice, but it is much more expensive than writing to an array cell.

6.5.3 Linked `pop`

```
bool pop(node **top, int *x) {
  node *old = *top;        /* no malloc! */
  if (old == NULL) return false;  /* empty */
  *x = old->data;
  *top = old->link;        /* advance top */
  free(old);               /* release memory */
  return true;
}
```

Pop also takes `node **top`, because it is going to advance the caller's `top` to the next node. The value removed is delivered through the output parameter `int *x`; the return value reports success. The body consists of these steps:

1. `node *old = *top;` saves the address of the top node in a local pointer. This line allocates no new memory; it only copies an address.
2. `if (old == NULL) return false;` reports underflow if the stack is empty.
3. `*x = old->data;` copies the value to the caller. This line must run before the node is released.
4. `*top = old->link;` advances the caller's `top` to the next node. This too must be done before the release, because the link field lives inside the node that is about to be released.
5. `free(old);` returns the node's memory to the system.

Apply four pops to the stack 30 → 20 → 10 and print each successful result. Table 6.7 summarizes the calls.

Call	`old`	`*x`	`*top` (after)	`free`	Returns	Stack
1st pop	node(30)	30	node(20)	node(30)	true	20 → 10
2nd pop	node(20)	20	node(10)	node(20)	true	10
3rd pop	node(10)	10	NULL	node(10)	true	(empty)
4th pop	NULL	(untouched)	NULL	—	false	(empty)

Table 6.7: Four pops from the stack 30 → 20 → 10.

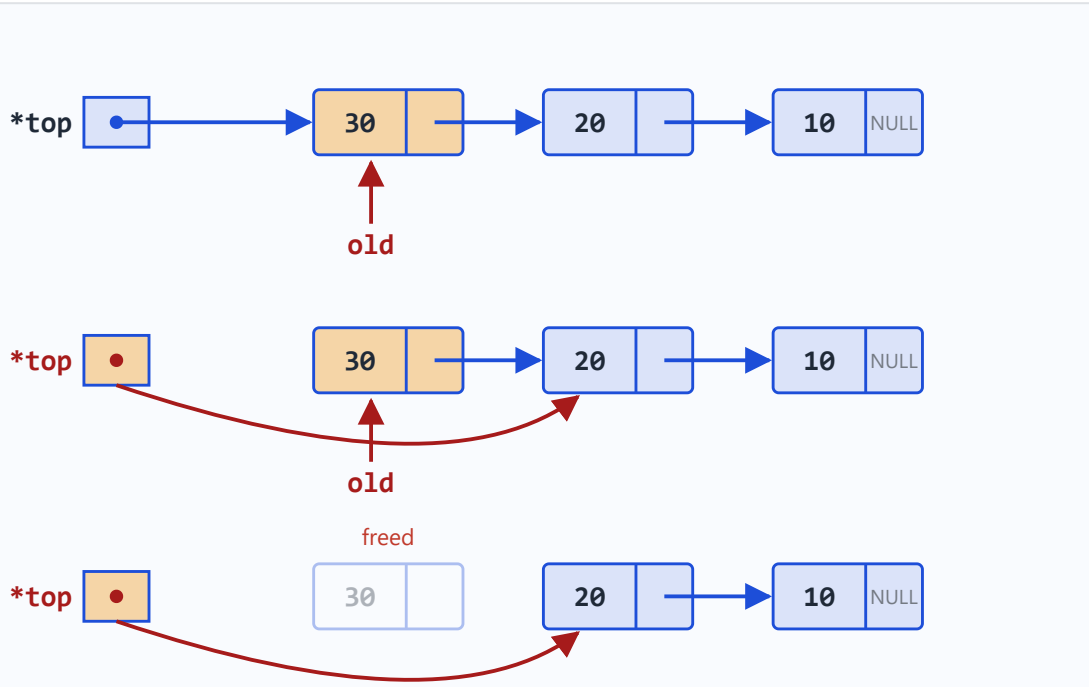


Figure 6.9: Three moments of the first pop. First, `old` points to the top node (30); no new memory has been allocated. Then, with `*top = old->link`, the top advances to 20; the node holding 30 is no longer in the list, but `old` still points to it. At the end, the node has been released with `free(old)`; `main` prints 30.

The test program pushed 10, 20, 30, called pop four times, and printed "30 20 10 false." The output order is the reverse of the input order. Pop also does a constant amount of work: one comparison, two assignments, and one `free`. Its cost is *O(1)*.

The order of the steps in pop matters even more than in push. After the call `free(old)`, the memory that `old` points to no longer belongs to the program. If `old->data` or `old->link` is read after that moment, the behavior is undefined. The program often reads the right value, because released memory is not handed to another use immediately. That makes the bug more dangerous: it does not

show up in tests, but the value is corrupted when another `malloc` intervenes. A pointer that points to memory released in this way is called a **dangling pointer**. The robust rule is this: first save the value and the link, and call `free` last.

6.5.4 Memory ownership and emptying the stack

In a linked stack, `malloc` allocates every node and one `free` must give each node back. In C, this responsibility lies entirely with the programmer. Nodes that remain on the stack when the program ends must be released too. A cleanup routine is written for this job:

```
typedef struct node {
    int data;
    struct node *link;
} node;

void stack_clear(node **top) {
    while (*top != NULL) {
        node *old = *top;
        *top = old->link;
        free(old);
    }
}                   /* afterwards *top == NULL */
```

`stack_clear` removes the top node until the stack is empty. Three lines run in each turn of the loop: the address of the top node is taken into `old`, `*top` is advanced to the next node, and then `old` is released. The order is again the same: a node is never released before its link is saved. When the loop ends, `*top` is `NULL`; the caller's `top` thus designates an empty stack and does not remain as a dangling pointer. In a stack of n nodes the loop runs n times; the cost is $\Theta(n)$. The extra memory is just the pointer `old`, that is, $O(1)$.

One point deserves notice in this code: `old` is a pointer variable, not a new node. A common mistake in student code is to allocate room with `malloc` for every pointer: `node *old = malloc(sizeof *old); old = *top;`. The first line allocates room for one node; the second line overwrites `old` with the address of the top node. The address of the allocated block is now nowhere. One node leaks at every pop. `malloc` is applied to a pointer only when the pointer is going to point to a new object. To point to an existing node, an assignment is enough.

That the linked stack runs without leaks was verified with a counting wrapper. The wrapper redirects calls to `malloc` and `free` to its own functions with the help of a macro and counts every successful call. The test program pushed 10, 20, 30 and popped four times, then pushed five more items, 1 through 5, and called `stack_clear`. The counter showed eight `malloc` calls and eight `free` calls; after the cleanup `top` was `NULL`. In the same program `sizeof(node)` came out as 16. The fields of the node are a 4-byte `int` and an 8-byte pointer; so that the pointer sits at an address that is a

multiple of 8, the compiler inserts 4 bytes of padding. For this reason, on a 64-bit system a linked stack uses four times as much memory per `int` as an array does; the bookkeeping records that `malloc` keeps for each block come on top of that.

Common mistake

Changing the top first in push: `*top = new_node; new_node->link = *top;`. When the second line runs, `*top` is already `new_node`; the node points to itself and all the nodes of the old stack are lost. Any loop that walks the stack from the top then spins forever on the same node. The correct order: first `new_node->link = *top;`, then `*top = new_node;`.

6.6 Array-based or linked?

So far two implementations have been built: the fixed-capacity array and the linked list. There is a third option in between: an array that grows itself when it fills up. This section first builds this third implementation, then justifies its cost with amortized analysis, and finally compares the three implementations.

6.6.1 The growable array

The problem with a fixed array is that the capacity must be known in advance. The solution is to keep the array in heap memory and, when it fills up, move it to a larger block with `realloc`. An array that is enlarged with `realloc` when full is called a **dynamic array**. The real question is how much to enlarge it. If the capacity is doubled at every fill-up, the average cost of the pushes stays constant. The implementation below follows the layout of the `struct` version:

```c
typedef struct {
    int *data;
    int top, cap;           /* top = -1 when empty */
} DynStack;

bool dyn_init(DynStack *s, int cap) {
    s->data = malloc(cap * sizeof *s->data);
    if (s->data == NULL) return false;
    s->top = -1;  s->cap = cap;
    return true;
}
bool dyn_push(DynStack *s, int x) {
    if (s->top == s->cap - 1) {          /* full: double the capacity */
        int *bigger = realloc(s->data, 2 * s->cap * sizeof *bigger);
        if (bigger == NULL) return false;  /* old block still valid */
        s->data = bigger;
        s->cap *= 2;
    }
    s->data[++s->top] = x;
    return true;
}
bool dyn_pop(DynStack *s, int *x) {
    if (s->top < 0) return false;
    *x = s->data[s->top--];
    return true;
}
void dyn_destroy(DynStack *s) {
    free(s->data);
    s->data = NULL;  s->top = -1;  s->cap = 0;
}
```

The `DynStack` structure carries three fields: the address of the array in heap memory, `data`; the top index, `top`; and the current capacity, `cap`. `dyn_init` allocates the array with the requested initial capacity. `dyn_pop` is identical to `pop` in the `struct` version.

The difference is in `dyn_push`. If the stack is full, the routine does not return an error; it tries to double the capacity. `realloc(s->data, new_size)` brings the old block to the new size. If there is room right behind the old block, the block can grow in place. If not, `realloc` allocates a new block, copies the old contents there, releases the old block, and returns the address of the new block. The number of items copied is the current capacity.

The result of `realloc` is not written directly to `s->data`; it is first taken into a temporary pointer named `bigger`. The reason is the failure case. If `realloc` fails, it returns `NULL` and leaves the old block alone. Had the result been written directly to `s->data`, the address of the old block would have been lost and all the items in it would have leaked. Thanks to the temporary pointer, on failure the stack remains valid as it was and the function returns `false`. On success, `data` and `cap` are updated, and then the item is written as usual.

When 10, 20, 30, 40, 50 are pushed onto a stack whose initial capacity is 2, the capacity changes as follows: 10 and 20 find room (`cap` = 2). When 30 is pushed the stack is full; the capacity rises to 4 and 2 items are copied. 40 finds room. When 50 is pushed the capacity rises to 8 and 4 items are copied. These values were verified by printing them in a test program: after each push, the pairs (`top`, `cap`) were (0, 2), (1, 2), (2, 4), (3, 4), (4, 8). The counting wrapper counted two `malloc` calls and two `free` calls.

6.6.2 Amortized cost

In a growable array, the cost of a single push varies greatly with the situation. Most pushes write to one cell and are $O(1)$. But the push made at the moment the capacity is full copies all the items and is $O(n)$. Worst-case analysis therefore says "push is $O(n)$." This is true but misleading: the expensive pushes are very rare. Finding the average cost per operation by dividing the total cost of a sequence of operations by the number of operations is called **amortized analysis**. The average here does not rest on probability; it holds even for the worst possible sequence of operations.

Theorem 6.1 (Amortized cost of doubling)

Perform n pushes in a row onto an empty growable-array stack of capacity 1, doubling the capacity each time it fills. The total number of items copied over all the enlargements is less than $2n$. Therefore the total cost of n pushes is $O(n)$ and the amortized cost per push is $O(1)$.

Proof. An enlargement happens when a push is made while the stack is full. Since the capacity goes 1, 2, 4, 8, ..., enlargements occur at capacities 1, 2, 4, ..., 2^k. An enlargement at capacity c copies c items. If the last enlargement happened at capacity 2^k, then at that moment the stack holds 2^k items and one more is being pushed; hence $2^k < n$. The total number of copies is $1 + 2 + 4 + \ldots + 2^k = 2^{k+1} - 1$. Since $2^k \le n - 1$, this total is at most $2n - 3$, a constant multiple of n. Each push also does one write. The total work is bounded by $n + (2n - 3) < 3n$, which is under 3 units per push. ■

The bound on the copies in the proof can be stated more tightly. The total of the copies is $2^{k+1} - 1$ and the final capacity is 2^{k+1}; that is, the total number of copies is one less than the final capacity. The final capacity is also less than $2n$. These numbers were verified with a test program: when 1000 pushes were made onto a stack of capacity 1, there were 10 enlargements, 1023 items were copied in total (fewer than $2n = 2000$), and the final capacity reached 1024.

The growth factor does not have to be two; what matters is that it is a constant factor greater than 1. If a factor of 1.5 is chosen, the copies increase somewhat and the idle capacity decreases; the amortized cost remains $O(1)$. By contrast, if the capacity is raised by a fixed amount at each fill-up, say 10 cells, the amortized cost breaks. Over n pushes there are about $n/10$ enlargements, and each copies all the current items. The total number of copies is $10 + 20 + \ldots + n \approx n^2/20$; the cost per push rises to $\Theta(n)$.

Common mistake

Writing `s->data = realloc(s->data, …);`. If `realloc` fails, it returns `NULL` and this assignment erases the only address of the old block. All the contents of the stack are lost and the old block leaks. The result must first be taken into a temporary pointer and written to the real field only after it is seen not to be `NULL`.

6.6.3 Comparison and measurements

Table 6.8 compares the three implementations on six criteria.

Criterion	Fixed array	Growable array	Linked list
Capacity	Fixed at compile time	×2 as needed	Until memory runs out
push / pop	*O(1)*	*O(1)* amortized	*O(1)*, a `malloc` per push
Extra memory per item	None	None (except unused capacity)	One pointer (8 bytes) and padding
Cache behavior	Very good	Very good	Poor (nodes scattered)
Overflow	Possible	Only if `realloc` fails	Only if `malloc` fails
Bug risk	Bounds checks	The result of `realloc`	Forgotten `free`, dangling pointer

Table 6.8: A stack as a fixed array, a growable array, and a linked list.

The "cache behavior" row needs an explanation. On every memory access, the processor brings into the **cache** not a single number but the 64-byte block around it. In an array stack, consecutive items lie side by side, so many items fit in one block and later accesses are served from the cache. In a linked stack, each node is a separate `malloc` block and where it will sit in memory is not known. Reaching consecutive nodes often requires fetching a new block.

The asymptotic costs are the same, but the constant factors differ greatly. To see this, a program that performs 10^7 pushes followed by 10^7 pops on each of the three implementations was compiled with `-O2` and run three times. The fixed array took 25–35 ms, the growable array (initial capacity 16) 30–63 ms, and the linked stack 680–806 ms. These values vary from machine to machine, but the ratio is consistent: the linked stack is roughly twenty times slower than the array stacks. Most of the difference comes from the `malloc` at every push and the `free` at every pop, and the rest from cache misses.

A practical rule for choosing is this. If the largest size the stack can reach is known, a fixed array is the simplest and fastest solution. If it is not known, a growable array is the fastest in practice. A linked stack suits cases where the nodes already live in other structures or where items are created and deleted one by one. For example, in a program whose items are large records that are shared with other lists, linking the nodes into the stack is cheaper than copying them into an array.

6.7 The call stack

The computer's most heavily used stack is not one that a programmer writes. Every C program uses a stack set up by the compiler and the operating system to manage function calls. It is called the **call stack**. This section explains how the call stack works, how it overflows, and how a recursive function can be written with our own stack instead of this one.

6.7.1 The frame

When a function is called, the information needed for the call to live is written to a block in memory. This block is called a **stack frame** (or **activation record**). The first part of the frame is the **return address**: it says from which instruction in the caller execution continues when the function ends. The frame also holds the function's parameters, its local variables, and the information needed to return to the caller's frame. A call pushes a new frame onto the call stack. `return` pops this frame and control passes to the return address. The function called last finishes first; this is why the call stack is a true stack.

A recursive function shows this arrangement clearly:

```
long fact(int n) {
    if (n <= 1) return 1;          /* base case */
    return n * fact(n - 1);        /* multiply when the call returns */
}
```

Let `main` call `fact(5)`. To compute its result, `fact(5)` waits for the result of `fact(4)`; `fact(4)` calls `fact(3)`, which calls `fact(2)`, which calls `fact(1)`. Each call has its own `n`, and these values do not get mixed up, because each lives in a separate frame. `fact(1)` reaches the base case and returns 1. At that moment the stack has the most frames: six frames, counting `main`. Then the frames are popped in reverse order: `fact(2)` returns 2·1 = 2, `fact(3)` returns 3·2 = 6, `fact(4)` returns 4·6 = 24, and `fact(5)` returns 5·24 = 120. Table 6.9 summarizes this process.

Moment	Call stack	Number of frames
`main` starts	main	1
`fact(5)` is called	main | fact(5)	2
`fact(4)` is called	main | fact(5) | fact(4)	3
`fact(1)` is called	main | fact(5) | fact(4) | fact(3) | fact(2) | fact(1)	6 (maximum)
`fact(1)` returns 1	main | fact(5) | fact(4) | fact(3) | fact(2)	5
`fact(2)` returns 2	main | fact(5) | fact(4) | fact(3)	4
`fact(5)` returns 120	main	1

Table 6.9: The call stack during `fact(5)` (the rightmost frame is on top).

That frames really are stacked in memory can be seen with an experiment. At each call of `fact`, the address of the parameter `n` was printed (`-O0`, Windows x86-64). At every call the address was 96 bytes lower than at the previous one: from a local variable in `main` to the `n` of `fact(5)` was 80 bytes, and to the `n` of `fact(1)` it was 464 bytes. These numbers vary with the compiler, the optimization level, and the processor. The observation that does not change is this: on x86 processors the call stack grows from high addresses toward low addresses, and each call adds a frame of fixed size. `fact(n)` therefore uses $\Theta(n)$ frames, that is, $\Theta(n)$ stack memory.

6.7.2 How the call stack overflows

The capacity of the call stack is limited and fixed when the program starts. Typical values are a few megabytes: 8 MB on Linux, 1 MB on Windows with the Microsoft compiler. The `zig cc` compiler used for this work allocated 16 MB to Windows programs. When the stack fills up, the program crashes with a **stack overflow** error. Below are two typical causes and one bug that comes from the same stack layout.

Infinite recursion
: A recursion with no base case, or one that never reaches it, pushes a frame at every call and pops none. For example, had the condition in `fact` been written as `n == 1`, the call `fact(0)` would have descended forever through $-1, -2, \ldots$. In an experiment, an infinite recursion that keeps a 64-byte local array in each call stopped at a depth of about 104,000 calls with the Windows error `STATUS_STACK_OVERFLOW` (0xC00000FD).

Large local arrays
: A local array such as `int a[1000000];` lives inside the frame and by itself takes 4 MB. It would overflow a 1 MB stack all on its own. In an experiment this array fit in a 16 MB stack, while an array of 5 million items (20 MB) crashed the program with the same error. Large data is kept in heap memory with `malloc` or defined as `static`.

Returning a local address
: Writing `return &local;` is not an overflow but a bug that comes from the same stack layout. When the function returns, its frame is popped. That region of memory is reused for the frame of the next call. The address returned now points to the local variables of another call; it is a dangling pointer.

Stack overflow must be kept in mind in every recursive routine in this course. For example, a function that frees a linked list recursively pushes one frame per node. For a list of a million nodes that means a million frames, and it can overflow the stack.

6.7.3 Turning recursion into a loop

Since the call stack is a stack, any recursive algorithm can be turned into a loop by doing, with our own stack, the work the call stack does. A recursive call puts "the work left unfinished for now" into a frame. In the loop version this unfinished work is pushed onto an explicit stack. For the factorial, the unfinished work is the information "multiply by `n` on return":

```
long fact_loop(int n) {
    Stack st;  int k;  long result = 1;
    stack_init(&st);
    while (n > 1)                  /* push the "calls" on our own stack */
        push(&st, n--);
    while (pop(&st, &k))            /* multiply on the "returns" */
        result *= k;
    return result;
}
```

The first loop imitates the descending calls of the recursive version: it pushes `n` until it falls to 1. For 5, the stack from bottom to top becomes 5, 4, 3, 2. The second loop imitates the returns: it multiplies the result by each value popped from the stack. The products are made in the order 2, 3, 4, 5, the same as the order of returns in the recursive version. The result is 120. `fact_loop(0)` and `fact_loop(1)` gave 1, and `fact_loop(10)` gave 3628800, the same as the recursive `fact`.

For the factorial this conversion is unnecessary; a simple `for` loop suffices. But in algorithms such as tree traversal, depth-first search, or backtracking, the unfinished work is not a single number. The unfinished work is a state, and that state is what gets pushed on the stack. Using our own stack has two benefits. First, our own stack can grow in heap memory; it is not bound by the few-megabyte limit of the call stack. Second, only the information that is needed is pushed onto the stack; there is none of the overhead of a frame, such as the return address and the saved registers.

6.8 Multiple stacks

A program can use more than one stack at a time. Giving each stack its own array is easy, but memory is wasted: when one stack fills up, the others may have free room that cannot be reached. This section deals with sharing several stacks in a single memory array.

6.8.1 Two stacks in one array

For two stacks there is a very neat solution. In an array of m cells, the first stack grows upward from cell 0 and the second grows downward from cell $m - 1$. The tops of the two stacks move toward each other.

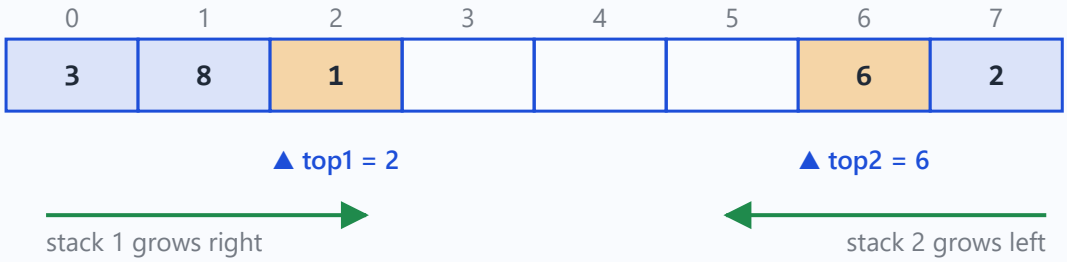


Figure 6.10: Two stacks in an array of eight cells. Stack 1 grows to the right from cell 0 and holds three items (3, 8, 1; `top1` = 2). Stack 2 grows to the left from cell 7 and holds two items (2, 6; `top2` = 6). The three cells in between are the free space shared by the two stacks.

When both are empty, `top1` = −1 and `top2` = *m*; each points to an imaginary cell outside the array. Pushing onto stack 1 increments `top1`, and pushing onto stack 2 decrements `top2`. The array is full when the two tops become adjacent, that is, when `top1 + 1 == top2`. The advantage of this arrangement is that no cell is wasted. Even if one stack holds few items and the other many, the array fills only when the total reaches *m* items. For three or more stacks this trick does not work, because an array has only two ends.

6.8.2 *n* stacks in one memory

For *n* stacks, the solution of Horowitz and Sahni is to divide the `memory` array of *m* cells into *n* segments. Each stack grows within its own segment. Two helper arrays are used. `top[i]` is the index of the top item of stack *i*. `boundary[i]` is the index of the cell just below the segment of stack *i*. Stack *i* uses the cells from `memory[boundary[i]+1]` to `memory[boundary[i+1]]`. To be able to write the upper bound of the last stack in the same way, `boundary[n]` = *m* − *1* is added.

Initially memory is divided into equal parts and all stacks are empty: for *i = 0, ..., n − 1*, `top[i] = boundary[i] = (m/n)·i - 1`. A stack is empty when its top equals its own lower boundary: `top[i] == boundary[i]`. It is full when its top reaches the lower boundary of the next stack: `top[i] == boundary[i+1]`. Push and pop are written with these two tests:

```
/* stack i: boundary[i]+1 .. boundary[i+1] */
bool push(int i, int x) {
  if (top[i] == boundary[i+1]) return false; /* full */
  memory[++top[i]] = x;
  return true;
}
bool pop(int i, int *x) {
  if (top[i] == boundary[i]) return false; /* empty */
  *x = memory[top[i]--];
  return true;
}
```

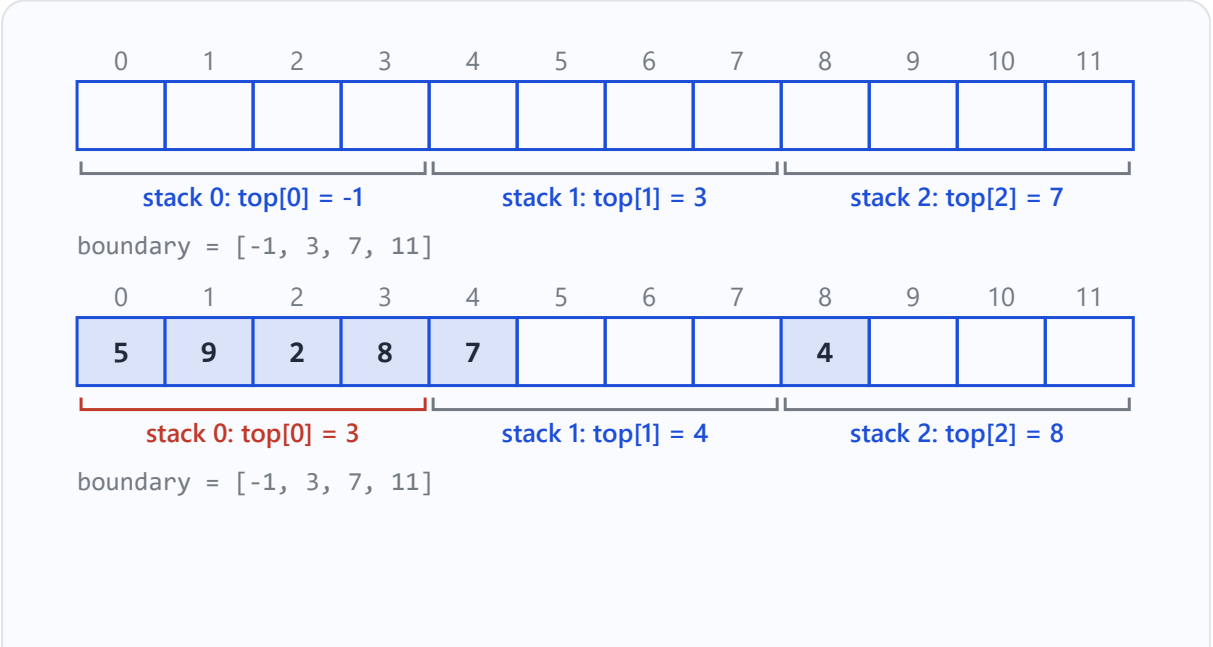
This code assumes that the arrays `memory`, `top`, and `boundary` are defined globally; it is kept short, as in the textbook. In a real program these three arrays are gathered in a `struct` and passed to the routines by address. The structure of the routines is the same as for a single stack; only the array `boundary` takes the place of the fixed limits. `push` returns `false` if the stack is full; otherwise it increments `top[i]` and writes the value. `pop` returns `false` if the stack is empty; otherwise it writes the top value to `*x` and decrements `top[i]`. Both are *O(1)*.

6.8.3 Code trace

Let $m = 12$ and $n = 3$. Each stack gets four cells. Initially `top` = [−1, 3, 7] and `boundary` = [−1, 3, 7, 11]. Stack 0 uses cells 0–3, stack 1 uses cells 4–7, and stack 2 uses cells 8–11. The calls `push(0, 5)`, `push(1, 7)`, `push(0, 9)`, `push(2, 4)`, `push(0, 2)`, `push(0, 8)`, `push(0, 6)`, and `pop(1, &x)` are made in turn. Table 6.10 gives each call.

Call	Test	Action	`top`
start	—	—	[−1, 3, 7]
`push(0, 5)`	−1 == 3? No	`memory[0] = 5`	[0, 3, 7]
`push(1, 7)`	3 == 7? No	`memory[4] = 7`	[0, 4, 7]
`push(0, 9)`	0 == 3? No	`memory[1] = 9`	[1, 4, 7]
`push(2, 4)`	7 == 11? No	`memory[8] = 4`	[1, 4, 8]
`push(0, 2)`	1 == 3? No	`memory[2] = 2`	[2, 4, 8]
`push(0, 8)`	2 == 3? No	`memory[3] = 8`	[3, 4, 8]
`push(0, 6)`	3 == 3? **Yes**	`false`: stack 0 is full	[3, 4, 8]
`pop(1, &x)`	4 == 3? No	`x = memory[4]` = 7	[3, 3, 8]

Table 6.10: Trace of multiple stacks with $m = 12$, $n = 3$; `boundary` = [−1, 3, 7, 11] does not change.



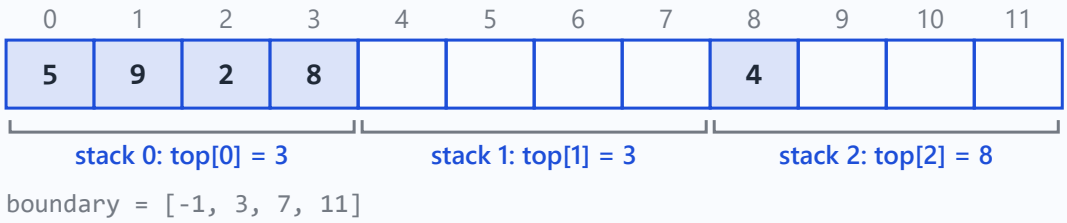


Figure 6.11: Three stacks in a memory of twelve cells. At the top, the start: all stacks are empty. In the middle, after the seventh call: the four cells of stack 0 are full (5, 9, 2, 8) and `push(0, 6)` is rejected, although six cells of memory are free. At the bottom, `pop(1, &x)` has removed 7; stack 1 is empty again.

The trace was verified with a test program; the `top` array after each call printed the same as in the table. The seventh call shows the weakness of the scheme. Stack 0 is full, because its top has reached the boundary of the next stack (`boundary[1]` = 3). Yet of the 12 cells in memory only six are occupied: stack 0 holds four items, stack 1 one, and stack 2 one. The remaining six cells are free. Even so, `push` says "full."

6.8.4 Making room and evaluation

Horowitz and Sahni solve this problem with a routine `stack_full(i)`. The routine is called when stack *i* fills up. First it looks among the stacks to the right for the first one with free room. If it finds one, it shifts the stacks in between one cell to the right and so opens one cell for stack *i*. If there is no room on the right, it makes the same search among the stacks on the left and shifts the ones in between, stack *i* itself included, one cell to the left. If there is no free cell anywhere, memory is truly full. Below, only the half that shifts to the right is given, simplified:

```
/* when stack i is full, look for room on the right; if found, shift
   the stacks in between one cell right (Horowitz's idea, simplified) */
bool make_room_right(int i) {
    int j = i + 1, k;
    while (j < N && top[j] == boundary[j + 1])   /* first stack with room */
        j++;
    if (j == N) return false;                    /* no room on the right */
    for (k = top[j]; k > boundary[i + 1]; k--)   /* memory[boundary[i+1]+1 .. top[j]] */
        memory[k + 1] = memory[k];               /* shift one right */
    for (k = i + 1; k <= j; k++) {               /* boundaries and tops shift too */
        boundary[k]++;
        top[k]++;
    }
    return true;
}
```

The function tries the stacks to the right of `i` in turn. It skips those with `top[j] == boundary[j+1]`, that is, those that are themselves full, and stops at the first stack with free room. If none is found, it returns `false`. If one is found, the cells from `boundary[i+1]+1` to `top[j]` are copied one cell to the

right. These cells hold the items of all the stacks from *i + 1* to *j*. The copying must start from the right; had it started from the left, each cell would have overwritten the next and values would have been lost. Finally both the lower boundaries and the tops of these stacks are incremented. The upper bound of stack *i*, `boundary[i+1]`, has also grown by one; stack *i* gains one cell.

Applied to the situation in the trace: stack 0 is full; the stack to its right, stack 1, has room (`top[1]` = 4, `boundary[2]` = 7). The only item of stack 1, 7, moves from `memory[4]` to `memory[5]`. `boundary[1]` rises to 4 and `top[1]` to 5. After that, `push(0, 6)` succeeds and 6 is written to `memory[4]`. A test program verified these steps: after making room, `top` = [3, 5, 8] and `boundary` = [−1, 4, 7, 11]; after the push, stack 0 held "5 9 2 8 6," stack 1 "7," and stack 2 "4."

The pros and cons of this approach are clear. Pros: a single array and a single `malloc` suffice and management is easy; push and pop are again *O(1)*; the same idea can be used to keep *n* queues in one array. Cons: when one segment fills up, a "full" error comes even though there is room in memory; shifting the stacks to make room is expensive. The number of items shifted can reach *m*. When memory is nearly full, every push may require a shift, and in the worst case every push is *O(m)*. It is also hard to estimate segment sizes in advance. If the stacks grow at different rates, equal division leads to frequent shifting.

6.9 Which output orders are possible?

To understand how a stack can change the order of its input, a classic question is asked. The numbers 1, 2, 3, 4 are to be pushed in this order. Any number of pops may be placed between the pushes, and at the end the stack is emptied. In what order can the numbers come out; that is, which permutations are possible as the **output order** of the pops?

Some orders are easy to get. For 1 2 3 4, a pop is made right after each push. For 4 3 2 1, four pushes are made, followed by four pops. For 2 4 3 1, this sequence is enough: push 1, push 2, pop (2), push 3, push 4, pop (4), pop (3), pop (1). But can 3 1 4 2 be obtained? If 3 is to come out first, 1 and 2 must have been pushed before 3 is pushed; at the moment 3 comes out, the stack holds 1 and 2 from bottom to top. The next item out can be either 2 on top or 4, which has not been pushed yet. Since 1 lies beneath 2, it cannot come out before 2. 3 1 4 2 is therefore impossible.

This reasoning turns into a general rule. At the moment an *x* comes out, all the items pushed before *x* and not yet popped are on the stack and must come out in the reverse of the order in which they were pushed. The rule can also be stated as a forbidden pattern.

Theorem 6.2 (Orders obtainable with a stack)

If 1, 2, …, *n* are pushed in this order, an output order $p_1\ p_2\ \ldots\ p_n$ can be obtained if and only if there are no three positions $i < j < k$ with $p_j < p_k < p_i$. In other words, the sequence contains no "large, small, medium" pattern.

Proof. First we show that the pattern is impossible. When p_i comes out, p_j and p_k, which are smaller than it, have already been pushed, since numbers are pushed in increasing order. Neither has come out yet, because they come after p_i in the output order. So both are on the stack, and since $p_j < p_k$, p_k was pushed later, that is, it lies above p_j. In this case p_j cannot come out before p_k; yet in the sequence it comes first. Contradiction.

For the converse, a greedy method is used: if the next target is p, push until p has been pushed, then pop the top item. The method gets stuck only if the target is not currently on top of the stack and has already been pushed. In that case, above the target p_j there is a larger item q that has not yet come out. q is at some position k in the sequence after p_j. All the items on the stack are smaller than the largest value that has come out so far, that is, than some p_i at an earlier position. This gives the pattern $p_j < q = p_k < p_i$. If there is no pattern, the method never gets stuck and the sequence is obtained. ■

In the sequence 3 1 4 2, the positions of 3, 1, 2 form this pattern: 3 is large, 1 is small, 2 is medium. So it is impossible. In 2 4 3 1 there is no such triple. The number of obtainable orders is also known. For n items this number is the nth **Catalan number**: $C_n = (2n)! / ((n + 1)! \cdot n!)$. The first values are 1, 2, 5, 14, 42, 132. For $n = 4$, only 14 of the 24 permutations are obtainable. For $n = 3$, only one of the six permutations, 3 1 2, is impossible. These numbers were verified for $n = 1, \ldots, 6$ by a script that tries every permutation with the greedy method; the same script also checked that for $n \le 7$ the pattern condition and the greedy method give the same result on every permutation.

Example 6.3

1, 2, 3, 4, 5 will be pushed in this order. Can the orders 3 5 4 2 1 and 5 3 4 1 2 be obtained?

3 5 4 2 1: push 1, push 2, push 3, pop (3); the stack is [1, 2]. Push 4, push 5, pop (5); the stack is [1, 2, 4]. Pop (4), pop (2), pop (1). Each target was on top at the moment it was popped; the order is obtained. Searching for the pattern also finds no "large, small, medium" triple.

5 3 4 1 2: for 5 to come out, all five numbers must have been pushed; when 5 comes out, the stack is [1, 2, 3, 4] from bottom to top. The next target is 3, but 4 is on top. The order cannot be obtained. As a pattern, the triple 5, 3, 4 (large, small, medium) is found.

6.10 Complexity summary

Table 6.11 gives the costs of the operations in this chapter for the three implementations side by side. Each value is justified in the relevant section.

Operation	Fixed array	Growable array	Linked
push	*O(1)*	*O(1)* amortized, worst case *O(n)*	*O(1)*
pop	*O(1)*	*O(1)*	*O(1)*

Operation	Fixed array	Growable array	Linked
peek / empty / full	*O(1)*	*O(1)*	*O(1)*
Memory for *n* items	MAX (fixed)	≤ *2n* cells	*n* · (item + pointer)
Clearing	*O(1)*	*O(1)* (a single `free`)	*Θ(n)*
Call stack of `fact(n)`	*Θ(n)* frames (independent of the implementation)		
Multiple-stack push	*O(1)*; making room *O(m)*	—	—

Table 6.11: Costs of the stack implementations (*n* is the number of items, *m* the memory size in multiple stacks).

The clearing row in the table stresses one difference. Array stacks sit in a single block; to clear a fixed array it is enough to set `top = -1`, and for a growable array a single `free` suffices. In a linked stack, however, each node is a separate block and `free` is called *n* times for *n* nodes. The bound "≤ *2n*" in the memory row holds when the capacity starts at 1 and is doubled; if the capacity is never shrunk, after many pops the idle capacity can remain far above *n*.

6.11 Common mistakes

Most of the bugs seen in stack code fall into a few patterns. The list below gathers the mistakes that can be met in each piece of code in this chapter.

1. **Skipping the bounds check.** A pop from an empty stack or a push onto a full stack reads or writes the array outside its bounds. C does not check this; neighboring memory is corrupted and the error shows up much later, in an unrelated place.
2. **Writing `*top--`.** The expression reads as `*(top--)` and decrements the pointer; the caller's top index does not change. The correct form is `(*top)--`. The compiler gives no warning.
3. **Confusing `top` with the item count.** `top` is the index of the last item; the item count is `top + 1`. The full test is `top == MAX - 1`, not `top == MAX`.
4. **Writing `*top = new_node` first in a linked push.** The address of the old list is lost and the new node points to itself.
5. **Reading the value after `free` in pop, or never releasing the node.** The first is undefined behavior through a dangling pointer; the second is a memory leak.
6. **Calling `malloc` needlessly on a temporary pointer.** `node *old = malloc(…); old = *top;` leaks a node at every call. An assignment is enough to point to an existing node.
7. **Writing the result of `realloc` straight into the real pointer.** On failure, the address of the old block is lost.
8. **Ignoring `bool` results.** The item of a failed push is silently lost; after a failed pop the old value goes on being used.

6.12 Exercises and solutions

The six questions on the exercise slide of the presentation are given below with their solutions. All the solutions were compiled and run with small test programs.

6.12.1 A `push` that returns an error

Question. Rewrite `push` of the global-array stack to return a `bool` instead of calling `exit`.

```
bool push(int *top, element item) {
    if (*top >= MAX_STACK_SIZE - 1)
        return false;          /* overflow: the caller decides */
    stack[++*top] = item;
    return true;
}
bool pop(int *top, element *item) {
    if (*top < 0)
        return false;          /* underflow */
    *item = stack[(*top)--];
    return true;
}
```

On overflow the function returns `false` and leaves the stack alone. If there is room, it writes the item and returns `true`. `pop` has to be written the same way; since the item can no longer be delivered as the return value, it is delivered through an output parameter, `element *item`. A test program that tried to push 10, 20, 30, 40 with capacity 3 printed "1 1 1 0": the fourth push was rejected. The loop that then emptied the stack gave 30, 20, 10, and `top` returned to −1. At no point did the program end through `exit`.

6.12.2 Two stacks in one array

Question. Write the routines `push(k, x)` and `pop(k, &x)` for two stacks in one array ($k = 1$ or 2).

```
#define M 10
int buf[M];
int top1 = -1, top2 = M;     /* both empty */

bool push(int k, int x) {
    if (top1 + 1 == top2) return false;   /* tops have met: full */
    if (k == 1) buf[++top1] = x;          /* stack 1 grows up */
    else        buf[--top2] = x;          /* stack 2 grows down */
    return true;
}
bool pop(int k, int *x) {
    if (k == 1) {
        if (top1 == -1) return false;     /* stack 1 empty */
        *x = buf[top1--];
    } else {
        if (top2 == M) return false;      /* stack 2 empty */
        *x = buf[top2++];
    }
    return true;
}
```

When both are empty, `top1` = −1 and `top2` = `M`. The full test is shared by the two stacks: `top1 + 1 == top2`. Pushing onto stack 1 is done with `buf[++top1]` and onto stack 2 with `buf[--top2]`; since stack 2 grows downward, its top decreases. In pop, the empty tests are separate: `top1 == -1` for stack 1 and `top2 == M` for stack 2. Popping from stack 2 is done with `buf[top2++]`.

With `M` = 10, seven pushes onto stack 1 and five onto stack 2 were tried. All seven of stack 1 went in; the first three of stack 2 went in and the next two were rejected, because the ten cells in total were full (`top1` = 6, `top2` = 7). That one stack held many items and the other few wasted no cell. All the operations are *O(1)*.

6.12.3 A linked stack that knows its size

Question. Move the linked stack into a `struct` that keeps a `size` field along with `top`.

```c
typedef struct {
    node *top;
    int size;
} SizedStack;

void sized_init(SizedStack *s) { s->top = NULL;  s->size = 0; }

bool sized_push(SizedStack *s, int x) {
    if (!push(&s->top, x)) return false;  /* reuse the linked push */
    ++s->size;
    return true;
}
bool sized_pop(SizedStack *s, int *x) {
    if (!pop(&s->top, x)) return false;
    --s->size;
    return true;
}
void sized_clear(SizedStack *s) {
    stack_clear(&s->top);
    s->size = 0;
}
```

`SizedStack` keeps the top pointer and the item count together. The routines reuse the linked stack's `push`, `pop`, and `stack_clear`; they pass them `&s->top`, that is, the address of the pointer field inside the structure. `size` changes only after successful operations: a failed push (out of memory) or a failed pop (empty stack) does not corrupt the counter. This way the number of items is known in *O(1)* without counting the list. After six pushes and one pop `size` was 5; after clearing it was 0, and the counting wrapper counted six `free` calls against six `malloc` calls.

6.12.4 Reversing a string with a stack

Question. Write a routine `void reverse(char *s)` that reverses a string using a stack.

Solution. This routine is the same as `reverse` traced in Table 6.5: the characters are pushed in order, then popped and written back into the same array. The time is $\Theta(n)$ and the extra memory is $\Theta(n)$. A robust version first finds the length of the string with `strlen` and creates the stack with that capacity (for example with `stack_create(n)`), so that the assumption about `CAP` is no longer needed.

6.12.5 The bottom item

Question. Write a routine that returns the bottom item of a stack using only push and pop; an auxiliary stack may be used.

```
/* if s is not empty, store the bottom item in *x; restore s */
bool bottom(Stack *s, int *x) {
    Stack aux;  int v;
    if (stack_empty(s)) return false;
    stack_init(&aux);
    while (pop(s, &v))           /* move everything to the auxiliary stack */
        push(&aux, v);
    *x = v;                      /* the last one out was the bottom */
    while (pop(&aux, &v))       /* move back: order is preserved */
        push(s, v);
    return true;
}
```

The routine moves all the items of the stack to the auxiliary stack. The item popped last during the transfer is the bottom item of the original stack; the variable `v` holds it when the loop ends. Then the items are moved from the auxiliary stack back to the original one. Since both transfers reverse the order, the original stack returns to its initial state. If the stack is empty to begin with, `v` would never be assigned, so the routine returns `false` at the very start. On a stack holding 4, 7, 1, 9 from bottom to top, the routine gave 4, and reading the stack from the top still gave 9, 1, 7, 4. In a stack of n items, $2n$ pushes and $2n + 2$ pops are made; the time is $\Theta(n)$ and the extra memory is $\Theta(n)$.

6.12.6 Number of frames in a `fact(5)` call

Question. What is the maximum number of frames on the stack during the call `fact(5)`? Draw them, counting `main`.

Solution. Since the base case is `n <= 1`, the calls are `fact(5)`, `fact(4)`, `fact(3)`, `fact(2)`, `fact(1)`. While `fact(1)` runs, these five frames and the frame of `main` are on the stack together: at most **6** frames. The drawing is the row "`fact(1)` is called" of Table 6.9: from bottom to top, main, fact(5), fact(4), fact(3), fact(2), fact(1). Had the base case been written as `n == 0`, `fact(0)` would be called too and the number would be 7.

6.13 Summary

- A **stack** is an ordered list in which insertion and deletion take place at the top end only. It is last in, first out (**LIFO**). Insertion is called **push**, removal **pop**, and reading without removing **peek**.
- Adding to a full stack is **overflow**, and removing from an empty stack is **underflow**. The stack ADT defines the operations and these error conditions independently of the implementation.
- In an **array stack**, `top` is the index of the top item: `top` = −1 means empty, `top` = MAX − 1 means full; the item count is `top + 1`. Push is written `stack[++*top] = item` and pop `return stack[(*top)--]`. `*top--` decrements the pointer and is wrong.

- Routines that return `bool` leave the decision to the caller, unlike routines that end with `exit`. The array and the top index are kept in a `struct`; no global variables are used. An **opaque type** in the header file hides the implementation from the code that uses it.
- In a **linked stack**, `top` is the head of the list; push and pop are *O(1)*. In push, link first and then change the top; in pop, save the value and the link first and call `free` last. Every `malloc` must have a `free`; the stack is emptied in *Θ(n)* with `stack_clear`.
- A **growable array** doubles its capacity with `realloc` when it fills up. A single push can be *O(n)*, but over *n* pushes the total number of copies is less than *2n*; the **amortized** cost of push is *O(1)*. In practice array stacks are much faster than a linked stack that calls `malloc` at every push.
- The **call stack** pushes a **frame** made of the return address, the parameters, and the locals for each call. Infinite recursion and large local arrays overflow it; returning the address of a local variable creates a dangling pointer. Every recursive algorithm can be turned into a loop that keeps its own stack.
- With **multiple stacks**, one memory array is shared through `top[i]` and `boundary[i]`. Two stacks growing from the ends waste no cells. With *n* stacks, when a segment fills up, room is made by shifting stacks; this can take *O(m)*.
- The output orders obtainable when 1, ..., *n* are pushed in this order are those that contain no "large, small, medium" pattern; their number is a Catalan number (14 for *n* = 4).

6.14 Key terms

stack last in, first out (LIFO) top push pop overflow underflow abstract data type (ADT) implementation interface header file include guard opaque type incomplete type global variable call by value operator precedence associativity const qualifier linked list node malloc free realloc memory leak dangling pointer dynamic array capacity amortized analysis cache call stack stack frame (activation record) return address stack overflow recursion base case balanced brackets expression evaluation postfix backtracking undo depth-first search (DFS)

6.15 Test Yourself

1. On a `struct` stack with `CAP` = 3, the following are applied in turn: `push 4`, `push 7`, `pop`, `push 1`, `push 6`, `push 2`; then the stack is emptied with the loop `while (pop(&st, &x))`. Which push returns `false`, and which values does the loop print?

   Answer

   4 and 7 go in; pop removes 7 and the stack is [4]. 1 and 6 go in; the stack is full with [4, 1, 6] (`top` = 2). So `push 2` returns `false`. The loop prints 6, 1, 4 and stops at the fourth pop.

2. In `pop`, `return stack[*top--];` has been written in place of `return stack[(*top)--];`. What does a loop that calls `pop` until the stack is empty do?

   Answer

   `*top--` reads as `*(top--)`: what is decremented is the function's local pointer, and the caller's `top` does not change. Every call returns the same top item and the stack never empties; the loop runs forever. The compiler does not report this bug with a warning.

3. 1, 2, 3, 4, 5 are pushed in this order. Which of the output orders 3 5 4 2 1 and 5 3 4 1 2 can be obtained?

   Answer

   3 5 4 2 1 can be obtained: push 1, 2, 3, pop (3), push 4, 5, pop (5), pop (4), pop (2), pop (1). 5 3 4 1 2 cannot: when 5 comes out the stack is [1, 2, 3, 4], and 3 lies beneath 4. As a pattern, the triple 5, 3, 4 is "large, small, medium."

4. On an empty linked stack, `push 10`, `push 20`, `push 30`, `pop`, `push 40` are applied. What does the stack look like from head to tail? How many `malloc` and how many `free` calls have been made so far? What is the total number of `free` calls after `stack_clear`?

   Answer

   The stack is 40 → 20 → 10 → NULL. Four pushes made four `malloc` calls and one pop made one `free`. `stack_clear` releases the remaining three nodes; the total number of `free` calls is 4, equal to the number of `malloc` calls.

5. 9 pushes are made onto an empty growable-array stack of capacity 1, doubling the capacity each time it fills. How many enlargements happen, how many items are copied in total, and what is the final capacity?

   Answer

   The enlargements happen at the 2nd, 3rd, 5th, and 9th pushes, at capacities 1, 2, 4, and 8: 4 enlargements. The number of items copied is 1 + 2 + 4 + 8 = 15 (less than $2n$ = 18). The final capacity is 16.

6. In the linked pop, the line `free(old);` is moved before the line `*x = old->data;`. The program

often works correctly in tests. Why is it wrong all the same?

Answer

After `free`, `old` is a dangling pointer and reading `old->data` is undefined behavior. Because released memory is not given to another use immediately, the old value often stays in place; that is why the tests pass. The value is corrupted when another `malloc` intervenes or when the allocator writes its own bookkeeping into the block.

7. With the multiple-stack structure for $m = 12$, $n = 3$ in its initial state, how many `push(1, ·)` calls in a row succeed? What does `pop(0, &x)` return in the meantime?

Answer

Stack 1 uses cells 4–7; four pushes succeed while `top[1]` goes from 3 to 7, and the fifth is rejected because `top[1] == boundary[2]` (7 == 7). Stack 0 is empty (`top[0] == boundary[0]` = −1); `pop(0, &x)` returns `false`.

8. Two stacks in one array (`M` = 10) start empty; 4 items are pushed onto stack 1 and 6 onto stack 2. What does the next `push(1, x)` return? Why?

Answer

`top1` = 3 and `top2` = 4. Because `top1 + 1 == top2` is true, the array is full and the call returns `false`. All ten cells are occupied.

9. If the capacity were raised by 10 cells each time instead of being doubled, what would the total cost of n pushes be?

Answer

There would be about $n/10$ enlargements, each copying all the current items: $10 + 20 + \ldots \approx n^2/20$. The total cost is $\Theta(n^2)$ and the amortized cost per push is $\Theta(n)$. For an amortized $O(1)$, the capacity must grow by a constant factor.

6.16 Further reading

- E. Horowitz, S. Sahni, and S. Anderson-Freed, *Fundamentals of Data Structures in C* (2nd ed.), Sections 3.1–3.2, 3.7, and 4.3: the source of the array stack, dynamic-array stack, multiple-stack, and linked-stack code these notes follow.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms* (3rd ed.), Section 10.1 and Chapter 17: defines the stack briefly in pseudocode; gives the amortized analysis of dynamic tables in detail with the aggregate, accounting, and potential methods.

- R. Sedgewick, *Algorithms in C, Parts 1–4*, Chapter 4: treats the stack in C as an example of an abstract data type, with the separation of interface and implementation.
- D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, Sections 2.2.1–2.2.2: studies sequential implementations of stacks, the shifting management of multiple stacks, and the permutations obtainable with a stack, with exercises.

6.17 References

1. E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2nd ed., Sections 3.1–3.2, 3.7, and 4.3.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 3rd ed., MIT Press, Section 10.1 (stacks) and Chapter 17 (amortized analysis).
3. R. Sedgewick, *Algorithms in C, Parts 1–4*, 3rd ed., Addison-Wesley, Chapter 4.
4. D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, 3rd ed., Addison-Wesley, 1997, Sections 2.2.1–2.2.2.

Queues

A stack inserts and removes at the same end, so the last item in is the first one out. Most lines in everyday life work the opposite way: at a bank counter, in a cafeteria, or at a printer, whoever arrives first is served first. A data structure that obeys this rule is called a queue. A new item enters at the rear, and the next item to be processed leaves from the front. These notes first define the queue as an abstract data type. They then build the simplest array implementation and use a code trace to expose its "false full" problem. The cure is the circular queue: the end of the array is treated as if it were joined to its beginning, and indices wrap around with the remainder operator. The remaining topics are three ways to tell a full circular queue from an empty one and the queue built on a linked list. The last section covers the double-ended queue, round-robin scheduling as used by operating systems, and how to build a queue from two stacks.

LEARNING OUTCOMES

The first-in-first-out rule
Define a queue as $Q = \{a_0, \ldots, a_{n-1}\}$, identify its front and rear ends, and list the operations of the queue abstract data type together with their error conditions.

The problem with the linear queue
Explain why cells vacated at the front of an array-based linear queue cannot be reused, and compute the cost of the shifting remedy.

Circular queue
Wrap indices with `% MAX`, compute the item count and a step backward correctly, and distinguish a full queue from an empty one in three ways.

Linked queue
Enqueue and dequeue in $O(1)$ time with `front` and `rear` pointers, handling the first insertion into an empty queue and the removal of the last item without memory errors.

Deque and applications
Implement a double-ended queue in a circular array, use a queue for time-sliced job scheduling, and build a queue from two stacks and justify its amortized cost.

7.1 What is a queue?

Stacks and queues are the two basic structures that deliberately restrict access to a collection of items. In both, only the item at one particular end can be reached; the items in the middle stay out of sight. This restriction is not a shortcoming. The narrower the access rule, the simpler and faster the structure is to implement, and the easier it is to predict how a program behaves. The only thing that separates the two structures is which item gets removed. In a stack it is the item inserted last. In a queue it is the item that has waited longest, that is, the one inserted first.

7.1.1 First in, first out

A line at a bus stop is the simplest example of a queue. A newcomer steps to the end of the line. When the bus arrives, the passenger at the head of the line boards. Cutting into the middle of the line or pulling someone out of it breaks the rule. Passengers board in exactly the order in which they joined the line, and the queue data structure reproduces this behavior faithfully.

Definition 7.1

A **queue** is an ordered list in which insertion takes place at one end and deletion at the other. In a queue $Q = \{a_0, a_1, \ldots, a_{n-1}\}$, the element a_0 is at the **front** and a_{n-1} is at the **rear**. For every i, a_i was inserted before a_{i+1}. A new element is inserted at the rear, and removal always takes place at the front. This rule is called **first in, first out** (**FIFO**).

The last sentence of the definition determines the entire behavior of a queue. If elements enter the queue in the order a_0, a_1, a_2, they leave in the same order. A stack given the same insertion order would produce the reverse removal order, a_2, a_1, a_0, which is why the stack rule is called **last in, first out** (**LIFO**). In short, a stack reverses the order and a queue preserves it.

Figure 7.1 shows the items A, B, C, and D being added to an empty queue in that order, followed by one removal. In an empty queue, front and rear are at the same place and there are no items between them. Each insertion moves the rear one cell forward. The removal returns A, which stands at the front because it entered first. Once A has left, the front cell is vacated and B becomes the new front. One detail in the figure deserves attention: removal does not move the remaining items. The front end advances one cell to the right while the items stay where they are. This detail is the source of every problem, and every solution, of the array-based queue.

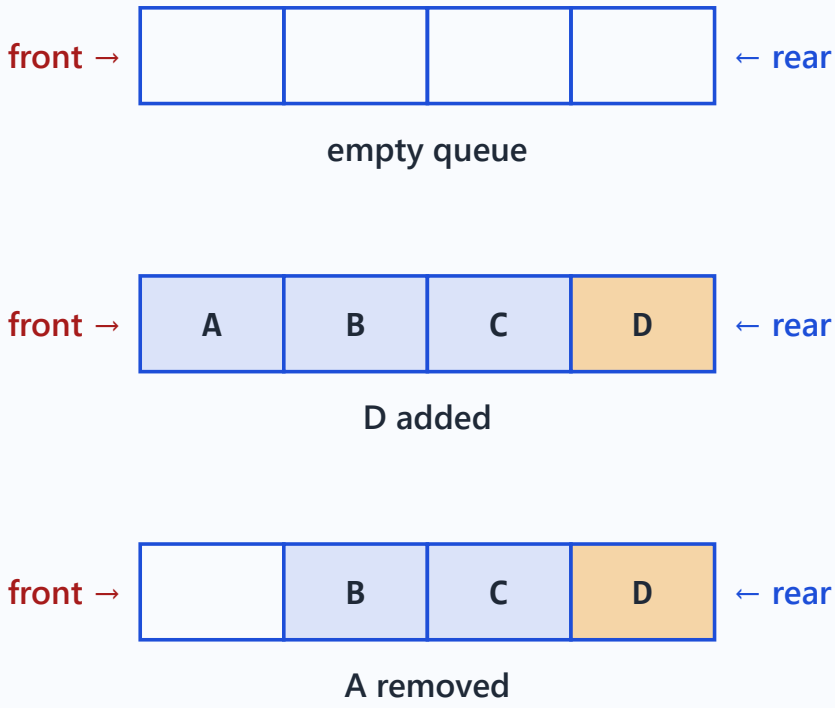


Figure 7.1: First in, first out. Top: the empty queue, with front and rear at the same place. Middle: A, B, C, and D have been added at the rear. Bottom: one removal has taken place and A, the first item added, has left the queue; the front cell is vacated and the other items have not moved.

7.1.2 The queue abstract data type

Before implementing a data structure, we must decide which operations will be performed on it and what each operation means. This specification is independent of the internal layout of the structure. A program that uses a queue need not know whether the items sit in an array or in a linked list; it only needs to know that insertion and removal obey the first-in-first-out rule. Giving the operations in this way, separately from any implementation, is called an **abstract data type** (**ADT**). Table 7.1 lists the six operations of the queue ADT.

Operation	Meaning	Error condition
`queue_create(max)`	Creates an empty queue that can hold at most `max` items.	—
`queue_full(Q)`	Returns true if no room is left in the queue.	—
`enqueue(Q, x)`	Adds `x` at the rear of the queue.	**Overflow** if the queue is full.
`queue_empty(Q)`	Returns true if the queue holds no items.	—
`dequeue(Q)`	Removes the front item from the queue and returns it.	**Underflow** if the queue is empty.
`queue_front(Q)`	Reads the front item without removing it.	Error if the queue is empty.

Table 7.1: Operations of the queue abstract data type.

The names of the two error conditions in the table are familiar from the stack. Trying to insert into a full structure causes an **overflow**, and trying to take an item from an empty one causes an **underflow**. The notion of "full" is meaningful only for fixed-size implementations: a queue built on a linked list does not fill up until memory runs out. Even so, the abstract data type includes the `queue_full` operation, so that the same interface covers both the array and the linked-list implementation.

The difference between a stack and a queue fits into one word in the table: end. A stack has a single active end, and both insertion and removal take place at the *top*. A queue has two ends, and each operation uses its own: insertion at the *rear*, removal at the *front*. The difference shows up in the implementation as well. A single index (`top`) suffices to manage a stack, whereas a queue needs two indices, one tracking the front and the other the rear. The position of these two indices relative to each other is the real subject of the rest of these notes.

7.1.3 Where queues are used

Queues appear wherever the principle "whoever's turn it is gets served" applies. Some of the uses are computer counterparts of real-life lines, and others sit inside the workings of algorithms.

- **Operating systems.** Processes that are ready to run wait in a queue for the processor, documents sent to a printer wait for the printer, and disk requests wait for the disk. It would be unfair if a later request could overtake the ones ahead of it; a queue provides this fairness automatically.
- **Buffers.** Keystrokes from the keyboard, packets from the network, and samples headed for the sound card wait in a **buffer**. A buffer is a queue that sits between the party that produces data and the party that consumes it. The two sides may run at different speeds: a keyboard can send several keys at once, and the program reads them a moment later. Such buffers are almost always fixed-size circular queues.
- **Breadth-first search. Breadth-first search** (**BFS**), which visits the nodes of a graph or tree in order of their distance from the start, rests on a queue. The neighbors or children of the node being processed are added at the rear of the queue, and the next node is taken from the front. In a tree the same method yields the nodes level by level, from left to right; this traversal is called **level-order** traversal.
- **Simulation.** When a bank counter, a traffic light, or a production line is simulated on a computer, every arriving customer or vehicle joins a queue. A simulation answers questions such as the average waiting time without building the real system.
- **Message queues.** In large software systems, servers pass jobs to one another through a message queue. Jobs are processed in the order in which they were sent, and they are not lost even if the receiver is busy at that moment.

One often confused structure must be added to this list. The name of the **priority queue** contains the word "queue", but it does not obey the first-in-first-out rule. In a priority queue every item has a priority, and each removal takes the item with the highest priority. The line in an emergency room is a

good example: a seriously ill patient goes ahead of mildly ill patients who arrived earlier. A priority queue is implemented efficiently with a tree structure called a heap. In these notes the word "queue", unless stated otherwise, means the ordinary queue that obeys the first-in-first-out rule.

The slides also contain an interactive queue. There the "add" button puts a new item at the rear of the queue and the "remove" button takes out the front item. After a few trials the same observation is confirmed over and over: the items come out in the same order in which they went in. For a stack, this order was exactly the reverse.

7.2 Linear queue in an array

The shortest way to build a queue in C is to use a fixed-size array and two integer indices. This section first gives that simplest implementation. It then uses a code trace to expose an important flaw, and examines the first, expensive way of removing the flaw.

7.2.1 Representation and C code

A single `top` index was enough for the stack. A queue has two ends, so two indices are kept: `front` and `rear`. The items lie side by side in the array, from the cell after `front` up to the cell that `rear` points to. The code below is the implementation in the book by Horowitz, Sahni, and Anderson-Freed:

```
#define MAX_QUEUE_SIZE 100
element queue[MAX_QUEUE_SIZE];
int front = -1;     /* one before the first item */
int rear = -1;      /* last item */

void enqueue(int *rear, element item) {
    if (*rear >= MAX_QUEUE_SIZE - 1)   /* any room? */
        queue_full();          /* warn and exit */
    queue[++*rear] = item;     /* bump, then store */
}
element dequeue(int *front, int rear) {
    if (*front == rear)
        queue_empty();         /* warn and exit if empty */
    return queue[++*front];    /* bump, return the front */
}
```

The first lines set up the memory of the queue. `MAX_QUEUE_SIZE` is the number of cells in the array. `element` is the type of the items stored in the queue. As in the stack implementation, this type is defined as a `struct` to suit the program; for example, it may carry a single `int value` field. The `queue` array and the two indices are global variables. Both indices start at −1.

The meaning of the two indices must be read carefully, because all of the code depends on it. `rear` points to the cell that holds the last item. `front`, however, points not to the first item but to the cell **one before the first item**. At first sight this choice looks odd, but it gives both operations the same shape: each first advances its own index and then uses that cell. The items of the queue therefore lie in cells `queue[front+1]`, …, `queue[rear]`, and the item count is `rear - front`. If the two indices are equal, the range is empty: **the condition for an empty queue is** `front == rear`. The initial values −1 and −1 satisfy this condition as well.

`enqueue` first checks whether there is room. If `rear` has reached the last cell of the array (`MAX_QUEUE_SIZE - 1`), no cell is left behind it and `queue_full` is called. If there is room, `++*rear` first increments `rear`, and the value of the expression is the incremented index; the item is written into that cell. `dequeue` first checks for emptiness. If the queue is not empty, `++*front` advances the front index to the cell of the item to be removed, and the item in that cell is returned.

The parameter types are chosen deliberately. Because `enqueue` changes `rear`, it takes it by address, as `int *rear`; the call has the form `enqueue(&rear, x)`. `dequeue`, on the other hand, changes `front` and only reads `rear`, so it takes `front` by address and `rear` by value. The parameter names are the same as the names of the global variables. This is legal in C: inside the function, the parameter name shadows the global variable. When reading the code, remember that `*rear` stands for the global `rear`, while a bare `rear` stands for the parameter's copy.

On an error, the code calls the functions `queue_full` and `queue_empty`. Like their counterparts in the stack implementation, these print a warning and terminate the program:

```
void queue_full(void) {
    printf("Queue is full!!!\n");  exit(-1);
}
void queue_empty(void) {
    printf("Queue is empty!!!\n");   exit(-2);
}
```

Terminating the program is crude error handling, but it keeps the code short and puts the main idea in the foreground at the first stage of learning. Once the queue is full or empty, the operation does not continue; therefore `enqueue` and `dequeue` need no separate branch for what happens after the error. Safe versions return `false` on error and leave the decision to the caller. That version is treated in the section on the circular queue.

Each operation performs one comparison, one increment, and one array access. Since a constant amount of work is done regardless of how many items the queue holds, the time cost of both operations is *O(1)*. The queue uses a fixed-size array; the memory cost is `MAX_QUEUE_SIZE` items and does not depend on the item count.

Common mistake

Writing `queue[*rear++]` instead of `queue[++*rear]`. Because the postfix `++` operator has higher precedence, the expression `*rear++` reads as `*(rear++)`. This advances the pointer itself rather than the integer that `rear` points to. The global `rear` does not change, and the item is written into the wrong cell. The prefix form `++*rear` first reaches `*rear` and then increments it. Wherever the precedence is in doubt, writing `++(*rear)` is the safest course.

7.2.2 Code trace: false full

The flaw of the linear queue becomes visible after a few steps of the code. In the trace the array is kept small: `MAX_QUEUE_SIZE = 4`. Initially `front = rear = -1`. The sequence of operations is as follows: A, B, and C are added; two items are removed; then D and E are added. Because the items are single letters, `element` can be thought of as `char` in the trace.

Operation	Check	`front`	`rear`	Array [0..3]	Result
Start	—	−1	−1	`. . . .`	Empty
`enqueue(&rear, 'A')`	−1 ≥ 3? No	−1	0	`A . . .`	`queue[0] = A`
`enqueue(&rear, 'B')`	0 ≥ 3? No	−1	1	`A B . .`	`queue[1] = B`
`enqueue(&rear, 'C')`	1 ≥ 3? No	−1	2	`A B C .`	`queue[2] = C`
`dequeue(&front, rear)`	−1 == 2? No	0	2	`. B C .`	returns A
`dequeue(&front, rear)`	0 == 2? No	1	2	`. . C .`	returns B
`enqueue(&rear, 'D')`	2 ≥ 3? No	1	3	`. . C D`	`queue[3] = D`
`enqueue(&rear, 'E')`	3 ≥ 3? **Yes**	1	3	`. . C D`	"Queue is full!!!", the program ends

Table 7.2: Trace of the linear queue with `MAX_QUEUE_SIZE = 4`. In the array column, a dot marks a cell that holds no queue item.

The first three insertions go smoothly. In each, the check says there is room, `rear` grows by one, and the letter is written into the new cell. In the two removals `front` advances first to 0 and then to 1, and A and B are returned in turn. What matters here is that removal does not change the array. A and B are still written in cells 0 and 1, but they are no longer part of the queue: the queue starts at `front+1`. These two cells are in effect empty.

When D is added, `rear` reaches 3, the last cell of the array. The queue holds only two items (C and D); the item count is `rear - front = 3 - 1 = 2`. Nevertheless, when E is to be added, the check gives `3 ≥ 3` and the code regards the queue as full. The program prints "Queue is full!!!" and ends, although two of the four cells are empty. This situation is called **false full**. Its cause is that both indices only move forward. As `front` advances, the cells it leaves behind are never used again. A linear queue can perform a total of `MAX_QUEUE_SIZE` insertions over its lifetime, no matter how many

items it holds at any one time. For example, if 60 items are added to a 100-cell queue and all 60 are removed, the queue is empty. Even so, at most 40 more insertions fit; the 41st insertion raises a "full" error.

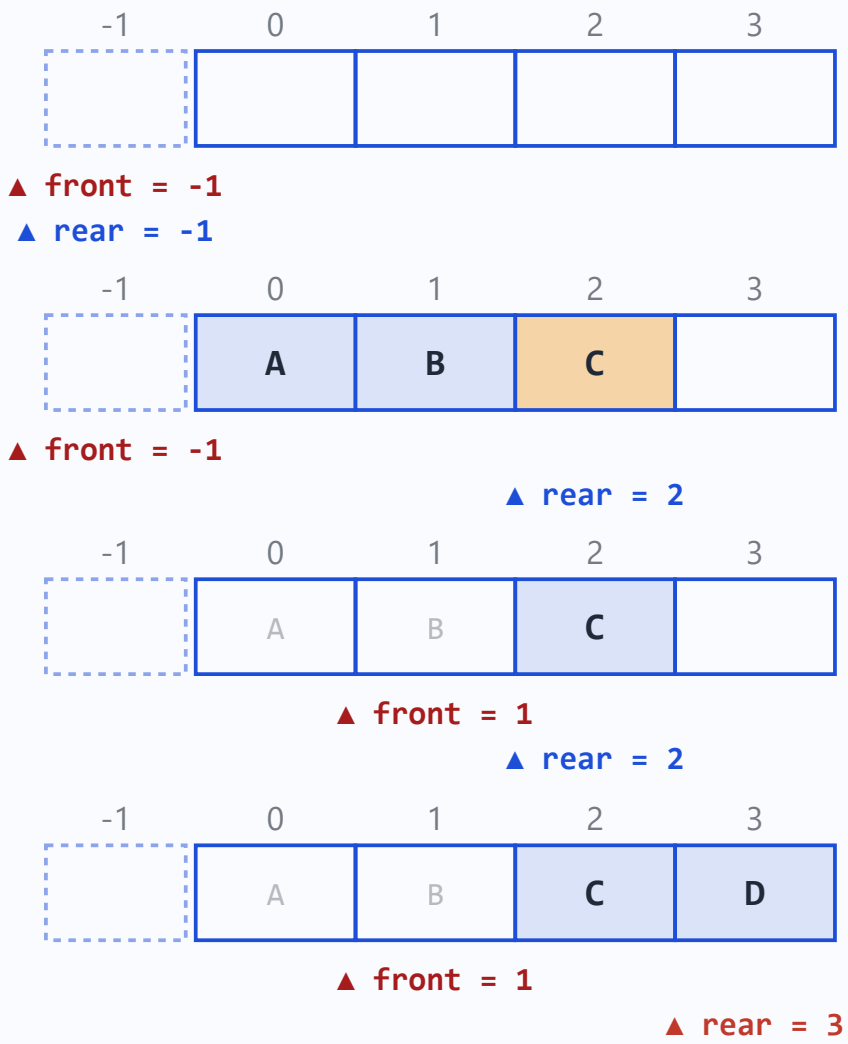


Figure 7.2: Four frames from the trace of the linear queue. At the start both indices are −1 (the dashed box on the left is position −1, outside the array). After A, B, and C are added, `rear = 2`. After two removals, `front = 1`; A and B are drawn faded because they no longer belong to the queue. When D is added, `rear = 3`, and the next insertion counts as full even though cells 0 and 1 are empty.

Common mistake

Mistaking the condition `rear == MAX_QUEUE_SIZE - 1` in a linear queue for true fullness. This condition only means "no room is left at the rear". The number of items actually in the queue is found with `rear - front`, and it may be very small. The queue is truly full only if `rear` has reached the end while `front == -1`.

7.2.3 Shifting: the expensive fix

The first fix that comes to mind for false full is to shift the items to the beginning of the array. When `rear` reaches the end and there are empty cells at the front, all items are moved to the left, `front` is pulled back to −1, and `rear` to one less than the item count. Figure 7.3 starts from the final state of the trace: `front = 1`, `rear = 3`, items C and D. After shifting, C moves to cell 0 and D to cell 1; `front = -1`, `rear = 1`, and two cells open up at the rear.

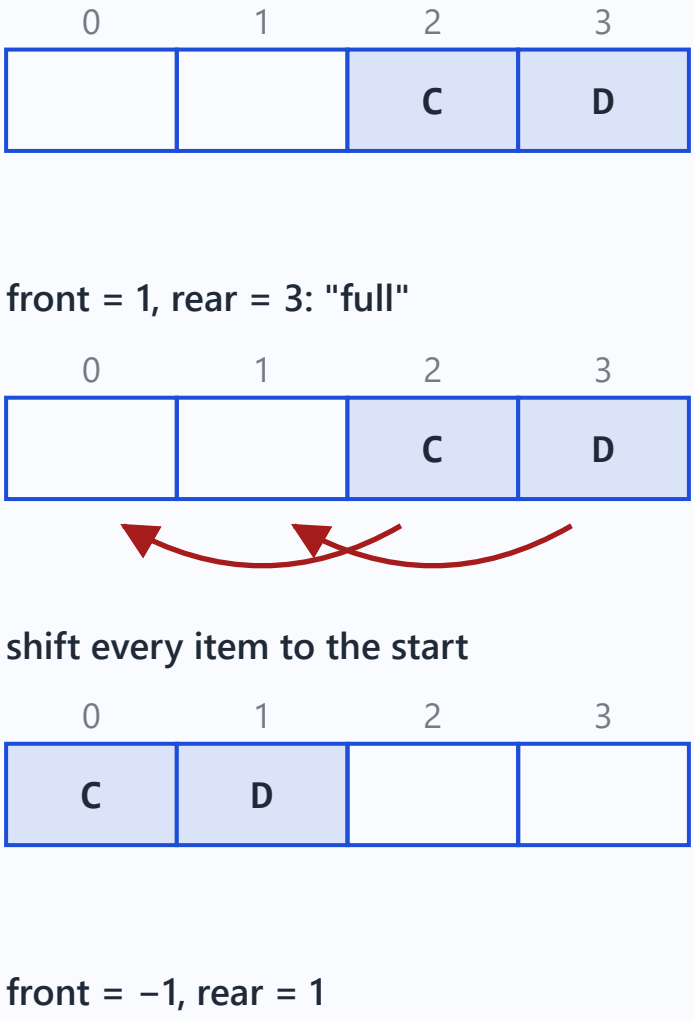


Figure 7.3: Shifting. Top: the state `front = 1`, `rear = 3` looks "full", although there are two empty cells at the front. Middle: every item is moved once toward the start. Bottom: `front = -1`, `rear = 1`, and room has opened up at the rear.

Adding shifting to the queue code is easy. The two functions below operate on the global `queue`, `front`, and `rear` of the previous section:

```
void shift_left(void) {       /* move items to the start */
    int n = rear - front;      /* item count */
    for (int i = 0; i < n; i++)
        queue[i] = queue[front + 1 + i];
    front = -1;
    rear = n - 1;
}
void enqueue_shift(element item) {
    if (rear == MAX_QUEUE_SIZE - 1) {
        if (front == -1) queue_full();  /* truly full */
        shift_left();                   /* room at front: O(n) */
    }
    queue[++rear] = item;
}
```

`shift_left` first computes the item count (`n = rear - front`). The loop moves the `i` -th item, the one in cell `queue[front+1+i]` , to `queue[i]` . The moves go from left to right; because the destination cell is always to the left of the source cell, no item that has not yet been read is overwritten. The last two assignments pull the indices to the new position. `enqueue_shift` comes into play only when `rear` has reached the end. If `front` is also −1 at that moment, there is no empty cell at the front and the queue is truly full. Otherwise it shifts first and then inserts the item as usual. Testing this version, the state in the trace (C, D; `front = 1` , `rear = 3`) becomes `front = -1` , `rear = 1` after shifting. E and F then fit, while G is rejected because the queue is truly full.

The solution is correct, but its price is high. Shifting moves each of the n items in the queue once, so its cost is $\Theta(n)$. Shifting is not performed at every insertion, only when `rear` has reached the end. However, if the queue runs almost full all the time (the normal situation for a buffer), nearly every insertion triggers a shift. Suppose the queue holds $M - 1$ items and one insertion and one removal are performed in succession. Each removal opens a single cell at the front. At the next insertion `rear` again hits the end, and $M - 1$ items are moved. Thus insertion goes from $O(1)$ to $O(n)$.

This estimate was also confirmed by measurement. With the queue holding $M - 1$ items at all times, 200,000 insertion–removal pairs were performed. For $M = 1000$ the total number of moves is 199,799,001, and for $M = 2000$ it is 399,798,001; that is, about M items were moved per insertion. The times are given in Table 7.15: the running time of the shifting queue grows in proportion to M, while that of the circular queue stays constant. In the slides this is put in the words of the instructor: shifting is the name of high cost in computer science. The way to escape shifting is to leave the items where they are and let the indices go around. This structure is called the **circular queue**.

7.3 Circular queue

The problem of the linear queue is that the indices stop when they reach the end of the array. The circular queue removes this limit: after the last cell, the index goes back to the first cell. The cells vacated at the front are thus used for items added at the rear, and no item is ever moved from its

place. This section takes up, in turn, the idea, the remainder operation that makes it possible, the code, and the problem of telling a full circular queue from an empty one.

7.3.1 Joining the end of the array to its beginning

Imagine an array of six cells as a ring instead of a flat strip. The cells, numbered 0 to 5, are the slices of the ring, and cell 5 is followed directly by cell 0 again. Nothing changes in memory; the array is still a flat block. Only the rule by which the indices advance changes. Figure 7.4 shows this ring: the queue holds A, B, and C, `front` points to 0, and `rear` points to 3.

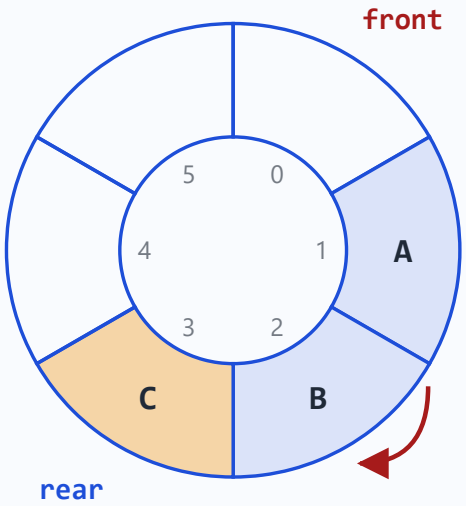


Figure 7.4: A six-cell circular array. The numbers on the inner ring are cell indices. `front = 0` and `rear = 3`; the queue holds A, B, and C in cells 1, 2, and 3. After cell 5 comes 0.

Definition 7.2

A **circular queue** is a queue implementation in an array of `MAX` cells in which the indices return to 0 after `MAX - 1`. The indices advance with `rear = (rear + 1) % MAX` and `front = (front + 1) % MAX`. As in the linear queue, `front` points to the cell one before the first item and `rear` points to the last item; initially `front = rear = 0`. The queue is **empty** when `front == rear` and **full** when `(rear + 1) % MAX == front`.

The full condition in the definition says: if advancing `rear` by one would land on `front`, there is no room. One consequence of this condition is that one of the `MAX` cells always stays empty. The queue holds at most `MAX - 1` items. Why this is so, that is, why the last cell is not filled, is the subject of the problem of telling full from empty. For now it is enough to note this: if the last cell were filled as well, `rear` would be equal to `front`. That is the same as the condition for an empty queue, and the code would mistake a full queue for an empty one.

7.3.2 Modular arithmetic and wrap-around

The tool that lets an index go around the ring is C's remainder operator `%`. The expression `a % m` gives the remainder of dividing `a` by `m`. For non-negative `a`, the result always lies between 0 and `m - 1`. Computing with numbers by their remainders modulo some *m* is called **modular arithmetic**. A clock is the most familiar example: three hours after 11 o'clock is not 14 but *(11 + 3) mod 12 = 2*.

In a circular queue the next value of `rear` is computed as `(rear + 1) % MAX`. If `rear` is not in the last cell, this expression is the same as the ordinary `rear + 1`. If `rear` is in the last cell, then `rear + 1 = MAX` and the remainder is 0: the index returns to the start. This event is called **wrap-around**. Table 7.3 gives four consecutive steps for `MAX = 6`.

`rear`	`rear + 1`	`(rear + 1) % 6`	**Meaning**
3	4	4	normal advance
4	5	5	last cell
5	6	0	**wrap-around**: back to the start
0	1	1	normal advance again

Table 7.3: Advancing of `rear` for `MAX = 6`.

There is a C-specific trap with the remainder operator. The C standard (since C99) truncates integer division toward zero. As a result, the remainder of a negative number is also negative: the value of `-1 % 6` is −1, not 5, and `-2 % 6` is −2. The mathematical *mod* operation always gives a non-negative result; C's `%` operator does not. This difference matters when an index has to be taken one step *backward*. The expression `(i - 1) % MAX` gives −1 for `i = 0`, and `queue[-1]`, outside the array, would be accessed. The correct form is `(i - 1 + MAX) % MAX`. Adding `MAX` keeps the value from being negative, and the remainder operation brings the result back into the range 0 to `MAX - 1`: `(0 - 1 + 6) % 6 = 5`. In a queue an index only moves forward; a step backward is needed in the double-ended queue, which works from both ends.

The second use of modular arithmetic is finding the item count. In the linear queue the item count was `rear - front`. In a circular queue `rear` can wrap around and fall behind `front`; then the difference is negative. The same correction works here:

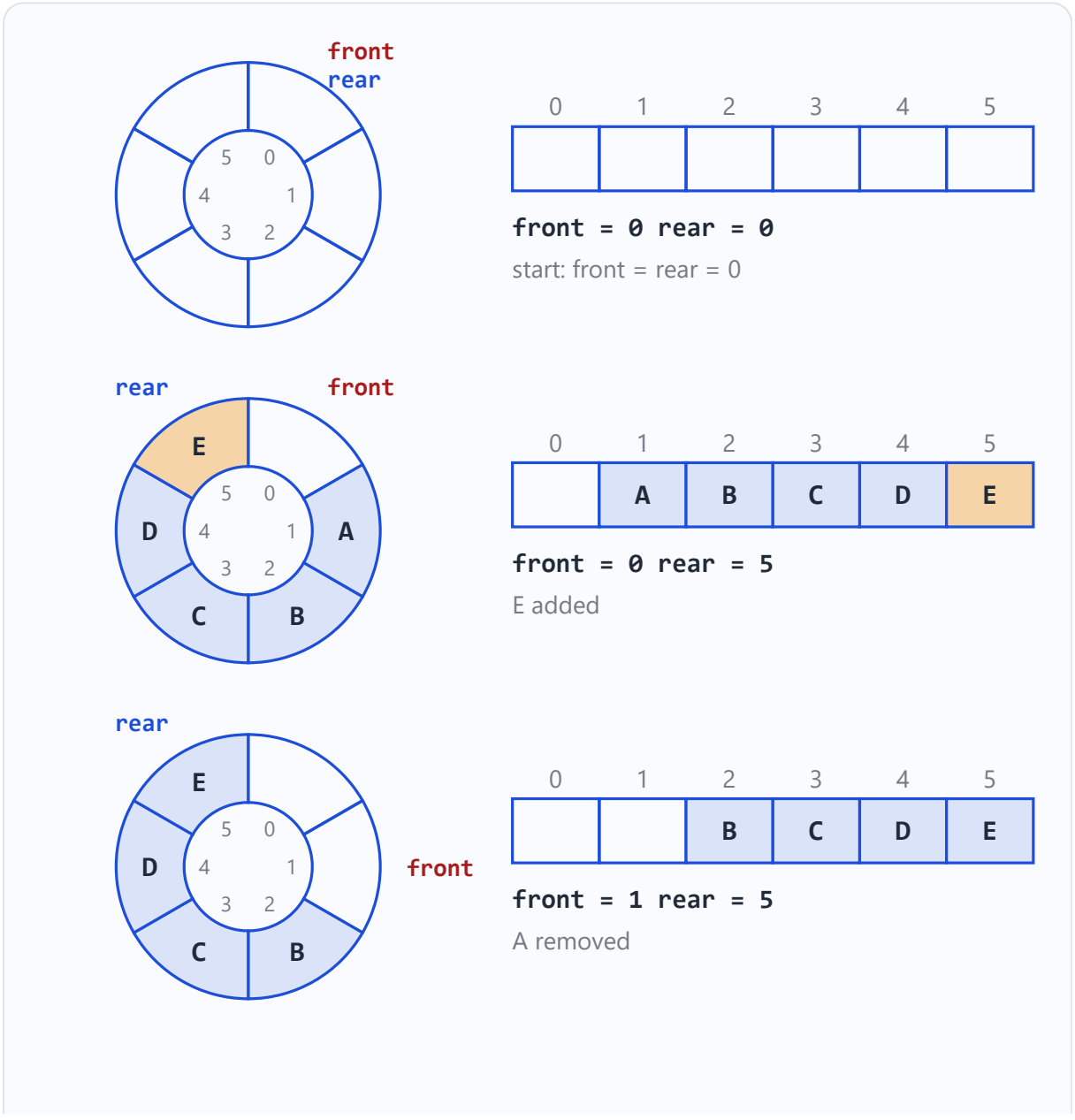
Example 7.1

Let `MAX = 6`, `front = 4`, and `rear = 1`. The item count is `(rear - front + MAX) % MAX = (1 - 4 + 6) % 6 = 3 % 6 = 3`. The items are counted from the cell after `front` up to `rear`, taking the wrap-around into account: cells 5, 0, and 1. The difference `rear - front` would give −3, which would be wrong. If `front` and `rear` are equal, the formula gives 0, which agrees with the condition for an empty queue.

7.3.3 Wrap-around with A…F

To see step by step how wrap-around works in a queue, consider a circular queue of six cells. Initially `front = rear = 0`. A, B, C, D, and E are added in turn. In each insertion, `rear` advances first and then the item is written into the new cell. This is why A goes into cell 1, not cell 0. After E is added, `rear = 5` and cell 0 is empty.

If one more insertion were attempted at this point, the value `(5 + 1) % 6 = 0` would equal `front` and the queue would count as full. Six cells hold five items; one cell is left empty on purpose. Next, A is removed: `front = (0 + 1) % 6 = 1` and A is returned. Now cells 0 and 1 are both empty. Then F is added. `rear = (5 + 1) % 6 = 0`, and F is written into cell 0, which was vacated at the front. The cell that the linear queue could not use is used here as if it lay at the rear. The queue is full again: `(0 + 1) % 6 = 1 = front`. The queue holds B, C, D, E, and F, and their order is preserved. The next removal returns B, because `front` advances from 1 to 2.



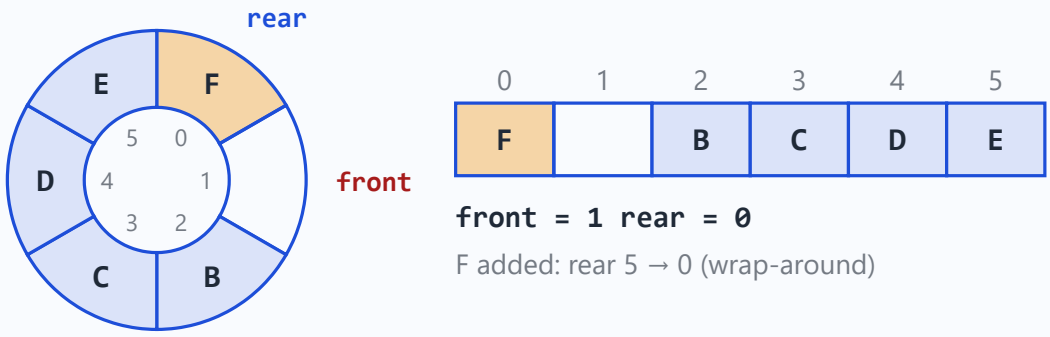


Figure 7.5: Wrap-around in a six-cell circular queue. Each frame shows the ring on the left and the flat view of the same array on the right. From top to bottom: the empty queue (`front = rear = 0`); A...E added (`rear = 5`); A removed (`front = 1`); F written into cell `rear = (5 + 1) % 6 = 0` . In the final state the queue is full: five items in six cells.

7.3.4 C code and a subtle point

The code of the circular queue differs from that of the linear queue in only two places: the indices advance with `% MAX_QUEUE_SIZE` , and the full condition changes. The version below again follows the implementation in the book by Horowitz, Sahni, and Anderson-Freed:

```
void enqueue(int front, int *rear, element item) {
    *rear = (*rear + 1) % MAX_QUEUE_SIZE; /* advance */
    if (front == *rear) queue_full();     /* full? */
    queue[*rear] = item;                  /* store */
}
element dequeue(int *front, int rear) {
    if (*front == rear) queue_empty();    /* empty? */
    *front = (*front + 1) % MAX_QUEUE_SIZE; /* advance */
    return queue[*front];                 /* return */
}
```

`enqueue` now takes `front` as a parameter as well, because the full condition depends on `front` . The function first advances `*rear` by one with wrap-around. If the advanced `rear` equals `front` , the queue is full and `queue_full` is called. Otherwise the item is written into the new cell. As in the linear version, `dequeue` first checks for emptiness, then advances `*front` with wrap-around and returns the item in that cell. Both functions perform a constant number of operations and move no item; the cost of each is *O(1)*.

The subtle point in this code is that `enqueue` advances `*rear` first and checks for fullness *afterward*. If the queue is full, `*rear` has already been set equal to `front` . In the version in the slides this does no harm, because `queue_full` terminates the program and the corrupted value is never used again. The problem appears when one writes a version that does not terminate on error but only prints a warning and returns. After an attempted insertion into a full queue, `front == rear` holds and the queue looks **empty**. All the items in it have vanished at once. This behavior was confirmed with a

small experiment: after A…E were added to a six-cell queue (`front = 0` , `rear = 5`), an insertion of F was attempted. After the warning was printed and the function returned, `rear = 0` and `front == rear` ; the queue looked empty.

The implementation in the book therefore restores `rear` to its old value when the queue is full. A cleaner way is to compute the new position first in a temporary variable and change `rear` only once it is certain that there is room. This approach is an instance of a general principle: **an operation that can fail must not modify the data structure when it fails**.

In the slides this slide appears next to a saying of the instructor. The saying says that studying at a good university is not a guarantee but a responsibility: a student should work knowing that failure means being expelled. The same attitude applies when writing code. It is not enough for code to "usually work"; edge cases such as insertion into a full queue must also be thought through deliberately.

7.3.5 The safe version and a code trace

The version below does not terminate the program. It returns `false` on error and `true` on success and leaves the decision to the caller. `dequeue` delivers the removed item not through its return value but through an output parameter (`element *x`), because the return value is reserved for the success flag. `<stdbool.h>` must be included for the `bool` type.

```
bool enqueue(int front, int *rear, element x) {
  int next = (*rear + 1) % MAX_QUEUE_SIZE;
  if (next == front) return false;      /* full */
  *rear = next;  queue[next] = x;
  return true;
}
bool dequeue(int *front, int rear, element *x) {
  if (*front == rear) return false;     /* empty */
  *front = (*front + 1) % MAX_QUEUE_SIZE;
  *x = queue[*front];  return true;
}
```

`enqueue` computes the new position in a local variable called `next` . If `next == front` , the queue is full and the function returns `false` without touching `rear` . If there is room, `*rear = next` is executed first and then the item is written. `dequeue` returns `false` on an empty queue. Otherwise it advances `front` , hands the item to the caller through `*x` , and returns `true` . The caller can catch the error with a check such as `if (!enqueue(front, &rear, x))` .

Example 7.2

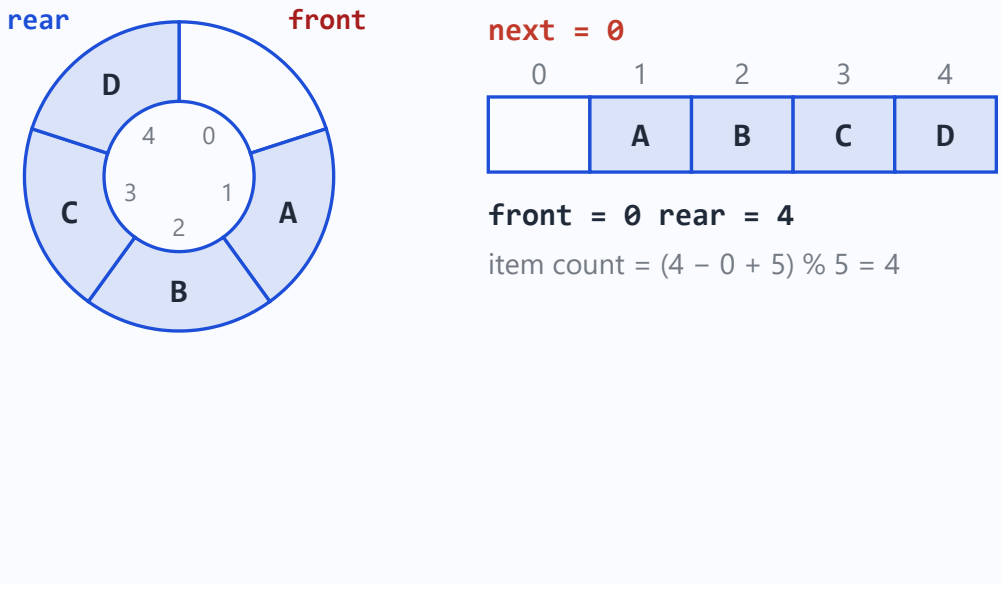
Let `MAX_QUEUE_SIZE = 5` and initially `front = rear = 0` . A, B, C, D, and E are to be added in turn; two items are removed; then E, F, and G are to be added. Table 7.4 gives the check and

the result of each call.

Call	Check	`front`	`rear`	Array [0..4]	Returns
Start	—	0	0	`. . . . .`	—
enqueue A	`next` = 1 ≠ 0	0	1	`. A . . .`	`true`
enqueue B	`next` = 2 ≠ 0	0	2	`. A B . .`	`true`
enqueue C	`next` = 3 ≠ 0	0	3	`. A B C .`	`true`
enqueue D	`next` = 4 ≠ 0	0	4	`. A B C D`	`true`
enqueue E	`next` = (4 + 1) % 5 = 0 = `front`	0	4	`. A B C D`	`false` (full)
dequeue	0 == 4? No	1	4	`. . B C D`	`true`, `*x` = A
dequeue	1 == 4? No	2	4	`. . . C D`	`true`, `*x` = B
enqueue E	`next` = (4 + 1) % 5 = 0 ≠ 2	2	0	`E . . C D`	`true` (wrap-around)
enqueue F	`next` = 1 ≠ 2	2	1	`E F . C D`	`true`
enqueue G	`next` = 2 = `front`	2	1	`E F . C D`	`false` (full)

Table 7.4: Trace of the safe circular queue (5 cells). In the array column, a dot marks a cell that holds no queue item.

Two rows of the table are especially important. At the first attempt to add E, the queue holds four items and one of the five cells (cell 0) is empty. Even so, `next` is 0 and equals `front`; the queue counts as full. The safe version leaves `rear` at 4 here. The queue is not corrupted, and the two removals return A and B in the correct order. At the second attempt to add E, `rear` wraps from 4 to 0, and E is written into cell 0, which a linear queue could never use. At the attempt to add G there are again four items (C, D, E, F), and the cell left empty, cell 2, is the cell that `front` points to. The removals that empty the queue from this state return C, D, E, F, which is consistent with the first-in-first-out rule.



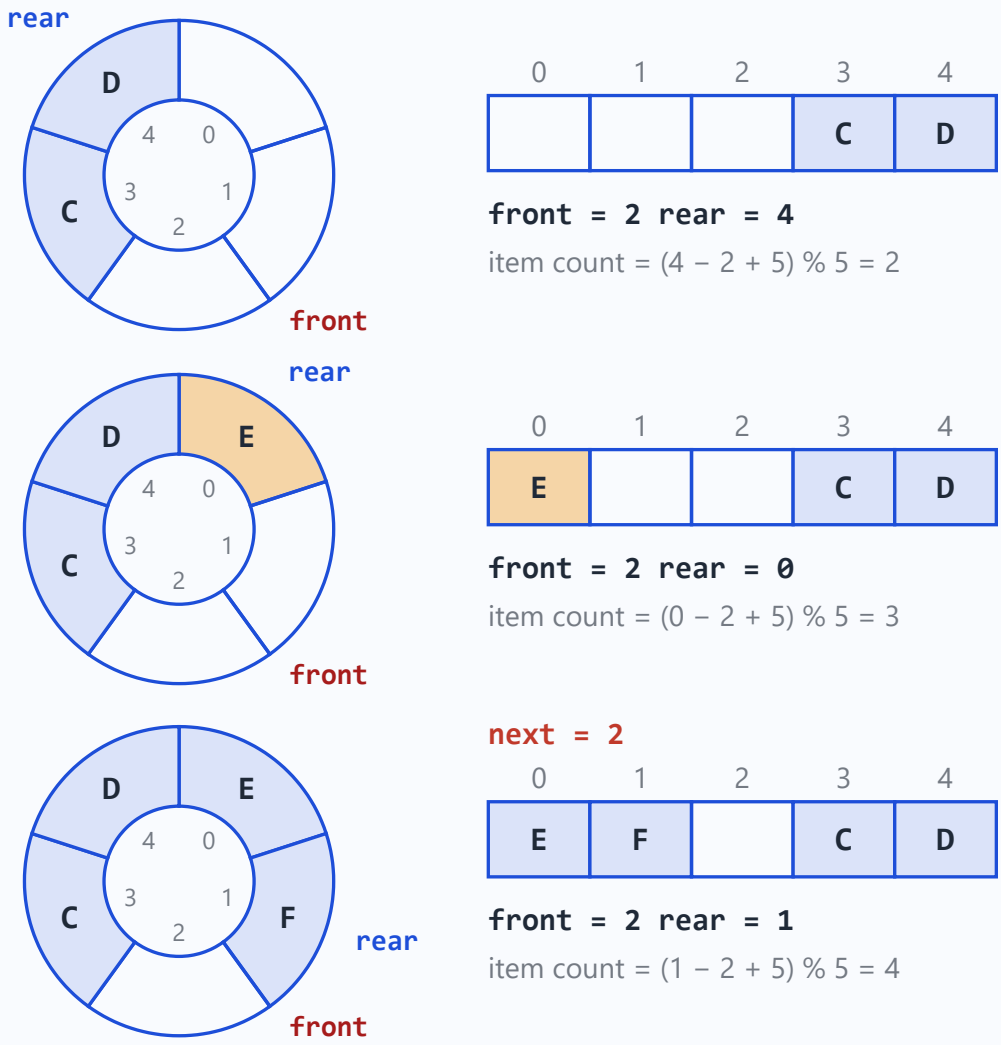


Figure 7.6: Four frames from the trace of the safe circular queue. From top to bottom: the attempt to add E after A...D (`next = 0 = front` , full); `front = 2` after two removals; E written into cell 0 by wrap-around; full again at the attempt to add G (`next = 2 = front`). In each frame the "item count" line applies the formula `(rear - front + 5) % 5` .

Common mistake

Forgetting `% MAX_QUEUE_SIZE` in one place in a circular queue, for example writing `*front = *front + 1` in `dequeue` . The program works correctly until the first wrap-around. When `front` takes the value `MAX_QUEUE_SIZE` after the last cell, an address outside the array is read. This is undefined behavior, and the bug often shows its symptoms only after a long run, in some unrelated place. Every line that advances an index must contain the remainder operation.

7.3.6 Full or empty? Three methods

In the previous two traces the queue was counted as full while one cell was still empty. This is not carelessness but a necessity. With `front` and `rear` alone, a full and an empty queue cannot be told apart. Counting is enough to express the state exactly.

> **Theorem 7.1 (Two indices are not enough)**
>
> If the state of a circular queue of `MAX` cells is represented only by the indices `front` and `rear`, it is impossible to distinguish all `MAX + 1` states in which the queue holds 0, 1, ..., `MAX` items. At least two different item counts map to the same index relationship.

Proof. The number of items in the queue depends only on the difference of the two indices: the items begin at the cell after `front` and extend up to `rear`. The difference is measured as `(rear - front + MAX) % MAX`, and this expression can take only the values 0, 1, ..., `MAX - 1`; that is, there are `MAX` different values. The item count, however, can take `MAX + 1` different values, from 0 to `MAX`. We must distribute `MAX + 1` states into `MAX` boxes. By the pigeonhole principle, at least one box receives two states. Concretely, the empty queue and the full queue with `MAX` items give the same difference (0): in both, `front == rear`. ■

The theorem leaves two ways out. Either one of the indistinguishable states is not used, or extra information is kept next to the indices. Table 7.5 compares three implementations of these two ways.

Method	Empty	Full	Capacity	Cost
1. Leave one cell empty	`front == rear`	`(rear+1)%MAX == front`	`MAX - 1`	One cell is wasted.
2. Keep a counter	`count == 0`	`count == MAX`	`MAX`	The counter is updated in every operation.
3. Keep a flag	`front == rear` and the last operation was a dequeue	`front == rear` and the last operation was an enqueue	`MAX`	The flag is updated in every operation.

Table 7.5: Three methods of distinguishing full from empty in a circular queue.

The first method is the one used by all the code so far. The state with `MAX` items is given up, so that `front == rear` describes only the empty queue. The cost is a single cell. If the items are large structures, this cell means a noticeable amount of memory, but in most applications it is negligible. The simplicity of the code makes up for the cost.

The second method keeps the item count in a separate variable, `count`. Empty and full are read directly from this counter, and all `MAX` cells are used. The counter has a further benefit: the item count is available at any moment and is read in *O(1)*. Moreover, with a counter there is no need to keep `rear`. The position of the last item is computed as `(front + count - 1) % CAP` and the next free cell as `(front + count) % CAP`.

The third method adds just one bit of information: whether the last operation was an insertion or a removal. The condition `front == rear` arises in only two ways. Either a removal has taken out the last item of the queue, or an insertion has filled the last empty cell. The flag tells these two possibilities apart. The code of the flag method is given in the exercises section.

7.3.7 Circular queue with a counter

The counter version gathers the whole state of the queue in a `struct`. Several queues can then be used in the same program, and no global variable is needed. The structure is defined as `typedef struct { int data[CAP]; int front, count; } Queue;`. In this version `front` points not to the cell before the first item but to **the first item itself**; an empty queue starts with `front = 0`, `count = 0`. The operations are as follows:

```
bool enqueue(Queue *q, int x) {
  if (q->count == CAP) return false;       /* full */
  q->data[(q->front + q->count) % CAP] = x;
  q->count++;
  return true;
}
bool dequeue(Queue *q, int *x) {
  if (q->count == 0) return false;         /* empty */
  *x = q->data[q->front];
  q->front = (q->front + 1) % CAP;
  q->count--;
  return true;
}
```

`enqueue` first checks for fullness: if `count == CAP`, there is no room. If there is room, the new item is written just after the last item, that is, into cell `(front + count) % CAP`, and the counter grows by one. `dequeue` returns `false` on an empty queue. Otherwise it copies the front item into `*x`, advances `front` by one with wrap-around, and decreases the counter by one. Both operations do constant work; their cost is *O(1)*. The functions take the queue through a `Queue *q` pointer. If the structure were passed by value, the function would work on a copy of it and the caller's queue would not change. The notation `q->count` is shorthand for `(*q).count`.

Example 7.3

Let `CAP = 3`, `front = 0`, and `count = 0`. The values 1, 2, 3, and 4 are to be added in turn; one item is removed; 4 is added; one more item is removed. Table 7.6 gives every step.

Call	Computation	front	count	data[0..2]	Returns
Start	—	0	0	`. . .`	—
`enqueue(&q, 1)`	pos = (0 + 0) % 3 = 0	0	1	`1 . .`	`true`

Call	Computation	`front`	`count`	`data[0..2]`	Returns
`enqueue(&q, 2)`	pos = (0 + 1) % 3 = 1	0	2	`1 2 .`	`true`
`enqueue(&q, 3)`	pos = (0 + 2) % 3 = 2	0	3	`1 2 3`	`true`
`enqueue(&q, 4)`	`count == CAP`	0	3	`1 2 3`	`false` (full)
`dequeue(&q, &x)`	`*x = data[0]`	1	2	`. 2 3`	`true`, `x` = 1
`enqueue(&q, 4)`	pos = (1 + 2) % 3 = 0	1	3	`4 2 3`	`true` (wrap-around)
`dequeue(&q, &x)`	`*x = data[1]`	2	2	`4 . 3`	`true`, `x` = 2

Table 7.6: Trace of the circular queue with a counter (`CAP = 3`).

This time all three cells fill up and no cell is wasted. The fourth insertion is rejected only because the counter has reached `CAP`. After one removal, the 4 that is added is written into cell 0 by wrap-around. In the final state the logical order of the queue, starting from `front`, is 3, 4: 3 is in the second cell and 4 is in the zeroth cell. The physical order in the array (4, ·, 3) differs from the order of the queue. In a circular queue the order is never read off the flat array; it is always read starting at `front` and wrapping around.

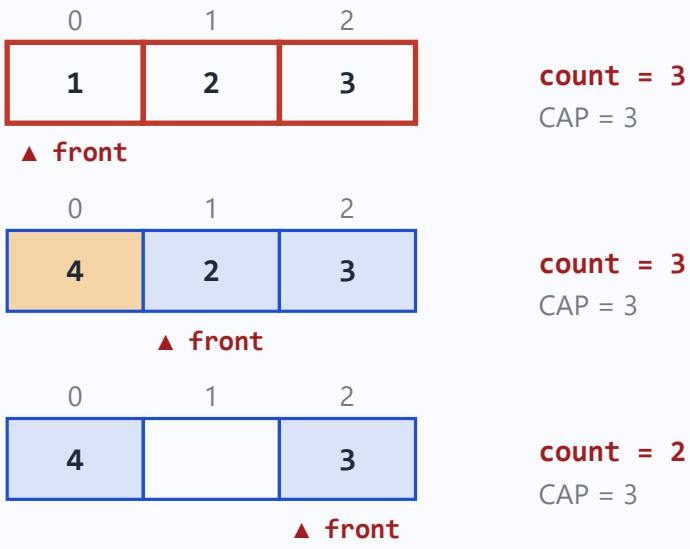


Figure 7.7: Circular queue with a counter (`CAP = 3`). Top: all three cells are full and the fourth insertion is rejected. Middle: 1 has been removed, and 4 has been written into cell 0 by wrap-around. Bottom: 2 has been removed as well: `front = 2`, `count = 2`.

Common mistake

Mixing up the two representations. In the array-based code of the slides, `front` points to the cell before the first item, and the operations work in the order "advance first, then use". In the counter version, `front` is the first item itself, and `dequeue` works in the order "use first, then

advance". If the code of one representation is used together with the initial values of the other, the queue skips the first item or returns an item that does not exist. In every implementation, what the indices point to should be written in a comment at the top of the code.

7.4 Linked queue

Every array-based queue has a capacity. The capacity is chosen when the program is written and does not change afterward. If it is not known in advance how many items will enter the queue, two bad options remain: reserve a large array and waste memory, or reserve a small array and risk an overflow. A linked list removes this dilemma. Each item lives in its own node; a node is allocated with `malloc` when needed and returned with `free` when it is done.

7.4.1 A queue on a linked list

Each node of a linked list carries a value and the address of the next node (the link). In these notes the node type is defined as `typedef struct node { int data; struct node *link; } node;`. The structure contains a pointer to its own type; this is why the name `struct node` is used before the `typedef` is complete. The link of the last node is `NULL`.

In a singly linked list, inserting at the head and deleting from the head are easy, because the first node is kept at hand. To insert at the end, one must reach the last node. If only the address of the first node is kept, the last node can be found only by walking the whole list from the beginning, which takes *O(n)* time. In a queue, insertion is made at one end and removal at the other, so both ends must be at hand. A linked queue is therefore represented by two pointers: `front` points to the first node and `rear` to the last. In an empty queue both are `NULL`.

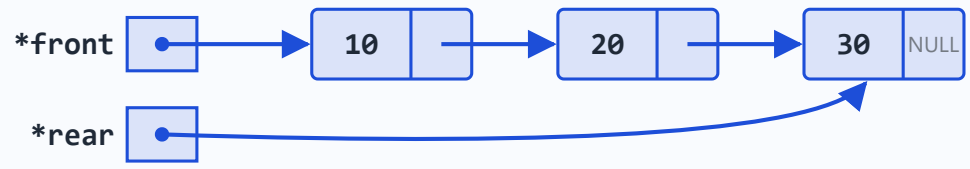


Figure 7.8: A linked queue made of 10, 20, and 30. `*front` points to the first node (10) and `*rear` to the last node (30). The link of the last node is `NULL`. The starred names remind us that the pointers are passed to the function by address.

Which end is the front and which is the rear is not chosen at random. Insertion is made at the *end* of the list: `rear->link = new_node; rear = new_node;` and the job is done. Removal is made at the *head* of the list: `front = front->link;` skips the first node. Both operations perform a constant number of assignments and are *O(1)*. If the opposite had been chosen, inserting at the head and

deleting at the tail, insertion would still be easy but deletion would become hard. To delete the last node, the link of the node before it has to be set to `NULL`. In a singly linked list one cannot go back from a node to its predecessor; the second-to-last node can be found only by walking from the head, in *O(n)* time. The rule in short: in a singly linked queue, **insert at the tail and delete from the head**.

This section relies on knowledge of pointers. In the slides, an anecdote of the instructor captures well the feeling students have about this topic. The instructor devoted two lectures to pointers. A student says that he knows pointers well but still does not understand what was explained. The instructor admits that this can happen. The safest way to grasp pointer code is to draw the boxes and arrows on paper after every assignment. The two traces below do this drawing step by step.

7.4.2 Code trace: `enqueue`

The insertion function may change both pointers of the queue. In the first insertion into an empty queue, both `front` and `rear` must point to the new node. For this reason both pointers are passed by address, that is, as pointers to pointers (`node **`). The call has the form `enqueue(&front, &rear, 10)`.

```
bool enqueue(node **front, node **rear, int x) {
  node *new_node = malloc(sizeof *new_node);
  if (new_node == NULL) return false;
  new_node->data = x;  new_node->link = NULL;
  if (*rear == NULL) *front = new_node; /* empty queue */
  else (*rear)->link = new_node;     /* link at end */
  *rear = new_node;
  return true;
}
```

The first line allocates memory for the new node. `sizeof *new_node` is the size of the type that `new_node` points to, that is, of `node`. Not repeating the type name keeps the code up to date if the type changes later. If `malloc` finds no memory, it returns `NULL`. In that case the function returns `false` without touching the queue. Then the fields of the node are filled in. Because the new node will be the last node of the queue, its link is set to `NULL`.

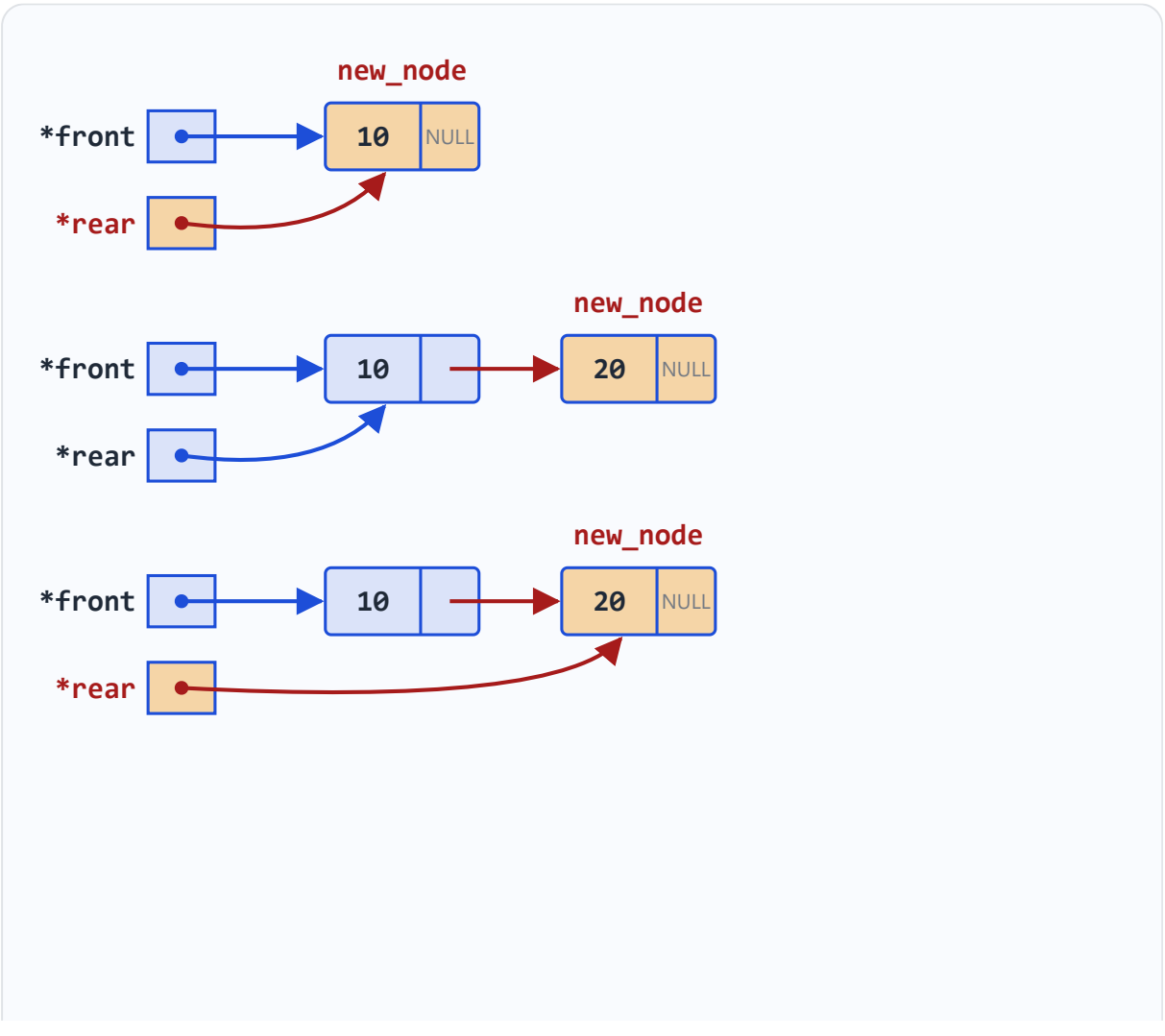
The next `if` separates two cases. If `*rear == NULL`, the queue is empty; the new node is also the first node, and `*front = new_node` is executed. If the queue is not empty, the new node is linked behind the old last node: `(*rear)->link = new_node`. The parentheses here are necessary. The `->` operator binds before `*`; had we written `*rear->link`, it would read as `*(rear->link)` and would not compile, because `rear` points to a pointer, not to a structure. In both cases the last line, `*rear = new_node`, moves the rear to the new node.

Call	Step	`*front`	`*rear`	Queue
Start	—	NULL	NULL	(empty)

Call	Step	`*front`	`*rear`	Queue
`enqueue(…, 10)`	`malloc` ; `data = 10` , `link = NULL`	NULL	NULL	(empty)
	`*rear == NULL` → `*front = new_node`	node(10)	NULL	10
	`*rear = new_node`	node(10)	node(10)	10
`enqueue(…, 20)`	`malloc` ; `data = 20` , `link = NULL`	node(10)	node(10)	10
	`(*rear)->link = new_node` : 10 now points to 20	node(10)	node(10)	10 → 20
	`*rear = new_node`	node(10)	node(20)	10 → 20
`enqueue(…, 30)`	`(*rear)->link = new_node` : 20 now points to 30	node(10)	node(20)	10 → 20 → 30
	`*rear = new_node`	node(10)	node(30)	10 → 20 → 30

Table 7.7: Trace of adding 10, 20, and 30 to an empty linked queue.

The trace shows that `front` changes only at the first insertion. In later insertions `front` always points to 10; what changes is the link of the old last node and `rear`. Each insertion performs one `malloc`, a few assignments, and one comparison. The length of the queue affects no step; insertion is $O(1)$. Had the `rear` pointer not been kept, the last node would have to be found by walking from the head at every insertion, and the total cost of n insertions would be $1 + 2 + \ldots + n = \Theta(n^2)$.



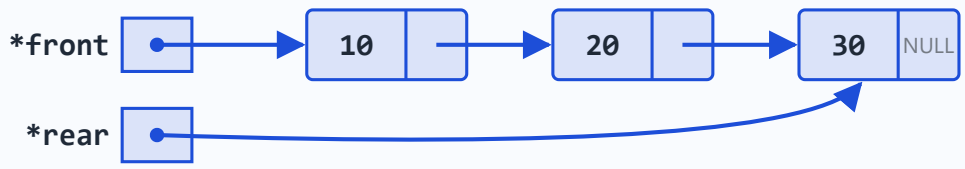


Figure 7.9: Four frames from the trace of `enqueue`. From top to bottom: after 10 is added, `*front` and `*rear` point to the same node; when 20 is added, the link of 10 is first turned to the new node (red arrow); then `*rear` moves to 20; the final state after 30 has been added as well.

Common mistake

Forgetting to update `front` at the first insertion into an empty queue, that is, skipping the `if` branch and writing `(*rear)->link = new_node` in every case. When the queue is empty, `*rear` is `NULL`, and writing to `NULL->link` crashes the program. If only the `NULL` check is added and `*front` is skipped, the program does not crash, but `front` stays `NULL`. The queue looks empty and the inserted items can never be removed; all of them leak.

7.4.3 Code trace: `dequeue`

The removal operation detaches the front node from the list, hands its value to the caller, and returns the node's memory:

```
bool dequeue(node **front, node **rear, int *x) {
  node *old = *front;
  if (old == NULL) return false;        /* empty */
  *x = old->data;
  *front = old->link;
  if (*front == NULL) *rear = NULL; /* last one gone */
  free(old);
  return true;
}
```

The function first saves the address of the front node in a local variable called `old`. This saving is mandatory. Once `*front` has advanced to the next node, there is no other way to reach the old node; if one advances without saving its address, the node cannot be freed and leaks. If `old` is `NULL`, the queue is empty and the function returns `false`. Otherwise the value is copied into `*x` and `*front = old->link` moves the front to the second node.

The next line is the easiest detail of the linked queue to forget. If the last item of the queue has been removed, `*front` is now `NULL`. At this moment `*rear` still points to the node that is about to be freed. Unless `*rear = NULL` is executed, `rear` becomes a **dangling pointer**, that is, a pointer to a memory region that is no longer valid. The next call to `enqueue` does not see `*rear` as `NULL`, does

not regard the queue as empty, and writes into the link field of the freed node. This is undefined behavior. Moreover, because `front` stays `NULL`, the new item is not visible in the queue either. Finally, `free(old)` returns the node. `free` must be called *after* the value has been read and the link followed; reading the fields of a freed node is also undefined behavior.

Call	Step	`old`	`*front`	`*rear`	Result
1. `dequeue`	`old = *front`	node(10)	node(10)	node(20)	—
	`*x = 10; *front = old->link`	node(10)	node(20)	node(20)	`*front ≠ NULL`: rear unchanged
	`free(old)`	—	node(20)	node(20)	`true`, 10
2. `dequeue`	`old = *front`	node(20)	node(20)	node(20)	—
	`*x = 20; *front = old->link`	node(20)	NULL	node(20)	`*front == NULL`
	`*rear = NULL; free(old)`	—	NULL	NULL	`true`, 20
3. `dequeue`	`old = *front = NULL`	NULL	NULL	NULL	`false` (empty)

Table 7.8: Trace of three calls to `dequeue` on the queue 10 → 20.

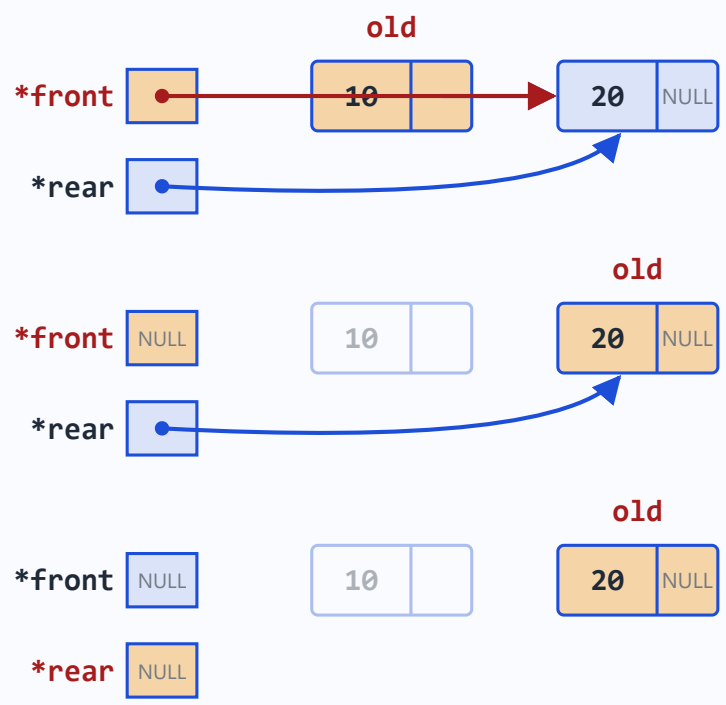


Figure 7.10: Three frames from the trace of `dequeue`. Top: after the first call `*front` has moved to 20; the node of 10 is held by `old`. Middle: in the second call `*front` has become `NULL`, but `*rear` still points to the node that is about to be freed. Bottom: `*rear = NULL` has been executed; the node can now safely be freed.

The two functions were tested together. When 10, 20, and 30 were added to an empty queue and all were removed, the values came out in the order 10, 20, 30, and after the last removal both `front` and `rear` were `NULL`. A new insertion (99) into the emptied queue correctly pointed both pointers to the new node. A test with 100,000 items also confirmed that the items came out in insertion order.

The `malloc` and `free` calls in the program were counted: every one of the 100,006 allocations was matched by a release, so there is no leak. On this compiler and platform a node is 16 bytes: a 4-byte `int`, 4 bytes of alignment padding, and an 8-byte pointer.

> **Common mistake**
>
> Not setting `rear = NULL` when the last item is removed from a linked queue. The bug does not show at once; it appears at the first insertion after the queue has emptied. The program sometimes seems to work and sometimes crashes in some unrelated place. The invariant of the two pointers in a linked queue is this: either both are `NULL`, or both point to a valid node.

7.4.4 Array or linked list?

The three implementations of the queue offer the same abstract operations, but their costs and constraints differ. Table 7.9 puts them side by side.

Criterion	Linear array	Circular array	Linked list
Enqueue / dequeue	*O(1)*, but "false full"	*O(1)*	*O(1)* (with `malloc` / `free`)
Is shifting needed?	Yes: *O(n)*	No	No
Capacity	`MAX` (total insertions over its lifetime)	`MAX - 1` (`MAX` with a counter)	As much as memory allows
Extra memory	None	None	One pointer per item
Special case	`rear` hits the end	Wrap-around and full/empty distinction	`rear = NULL` when emptied

Table 7.9: Comparison of queue implementations.

The linear queue serves teaching purposes only; it is not used in real programs. The choice between the circular array and the linked list depends on the application. The circular array uses memory as a single block. Because the items sit side by side in memory, it makes good use of the processor cache and does not call `malloc` in every operation. A linked queue needs no upper bound, but it carries a pointer for every item and every insertion is a memory allocation call. In a queue of `int` this means 16 bytes per item, where an array needs 4.

In practice two structures are used most: the circular array for fixed-capacity buffers, and the **growing circular array** for queues whose upper bound is unknown. In the growing version, when the array becomes full, a new array of twice the size is allocated and the items are moved there. The move cannot be done with `realloc`. `realloc` copies the old block as it is; if the queue has wrapped around, some of the items are at the end of the array and some at the beginning, and in the enlarged array empty cells would remain between them. The items are therefore copied one by one in logical order, starting at `front`.

```c
typedef struct {
  int *data;
  int cap, front, count;
} DQueue;

bool grow(DQueue *q) {
  int new_cap = 2 * q->cap;
  int *bigger = malloc(new_cap * sizeof *bigger);
  if (bigger == NULL) return false;
  for (int i = 0; i < q->count; i++)              /* copy in queue order */
    bigger[i] = q->data[(q->front + i) % q->cap];
  free(q->data);
  q->data = bigger;  q->cap = new_cap;  q->front = 0;
  return true;
}
bool dq_enqueue(DQueue *q, int x) {
  if (q->count == q->cap && !grow(q)) return false;
  q->data[(q->front + q->count) % q->cap] = x;
  q->count++;
  return true;
}
```

`grow` allocates the new array, takes the `i`-th item from cell `data[(front + i) % cap]` and writes it into the `i`-th cell of the new array, releases the old array, and pulls `front` back to 0. `dq_enqueue` grows the queue only when it is full; if growing fails, it returns `false` without touching the queue. For example, suppose 1…4 are added to a four-cell queue, two items are removed, and then 5 and 6 are added. The array physically contains 5, 6, 3, 4, and `front = 2`. When 7 is added, the array grows to eight cells and the items are laid out from the start in the order 3, 4, 5, 6; 7 goes into the fifth cell. A copy that disturbed the order would make the queue return some order other than 3, 4, 5, 6, 7.

Growing by itself takes $\Theta(n)$ time, but it is done rarely. Because the capacity doubles each time, the total number of items copied over n insertions is $1 + 2 + 4 + \ldots < 2n$. The average cost per insertion therefore stays constant; insertion is **amortized** $O(1)$. In the experiment, one million items were added to a queue whose capacity started at 1: 20 growth steps were made, a total of 1,048,575 items were copied, and this number stayed below $2n$. All items came out in insertion order; all 23 `malloc` calls were matched by `free`.

A linked queue is the natural choice if the items already live as nodes. For example, an operating system strings its process descriptor structures into a queue using their own link fields. In this case the queue allocates no separate memory; it merely arranges the links that already exist.

7.5 Deques and applications

Queues and stacks restrict access to one or two ends. This section first introduces the general structure in which both ends are open for both insertion and removal, the double-ended queue. It then moves on to three applications of the queue: keeping several queues in a single array, time-sliced scheduling in operating systems, and building a queue from two stacks alone.

7.5.1 The double-ended queue

Definition 7.3

A **double-ended queue**, or **deque** for short (pronounced "deck"), is an ordered list in which insertion and deletion can take place at both ends. It has four basic operations: insert at the front, insert at the rear, remove from the front, and remove from the rear. In a suitable implementation all four take *O(1)* time.

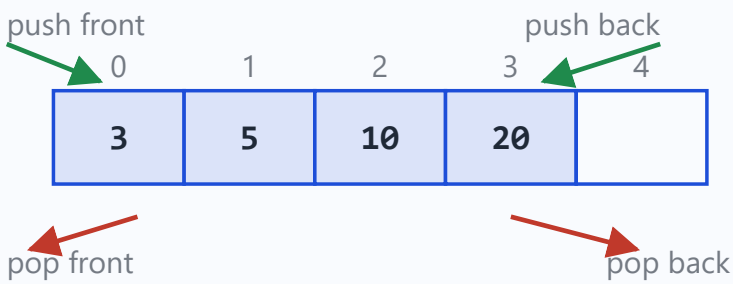


Figure 7.11: The four operations of a deque. Green arrows show insertion and red arrows show removal; both ends are open to two operations.

The deque is a general structure that covers both the stack and the queue. If only one end is used, for example always inserting at the rear and always removing from the rear, the deque behaves like a stack: after 1, 2, 3 are inserted, 3, 2, 1 come out. If items are inserted at one end and removed from the other, it becomes a queue: the same insertion order gives the removal order 1, 2, 3. Both behaviors were checked by testing the implementation below. There are also two restricted kinds. In an **input-restricted** deque, insertion is allowed at one end only, while deletion is allowed at both. In an **output-restricted** deque, insertion is allowed at both ends and deletion at one end only.

The deque is useful in many algorithms. In the **sliding window** problem, a window of fixed width is slid along an array and the largest value in the window is wanted at every position. A deque holds the indices of the candidate values. The index that leaves the window is dropped from the front, and candidates smaller than the newcomer are dropped from the back. The whole array is thus processed in *O(n)* time. In **work stealing**, a method used in multicore schedulers, each core keeps its own jobs in a deque. The core takes its own jobs from one end, while an idle core "steals" a job from the other

end. The undo history of a text editor can also be kept in a deque. New operations are added at the rear and undo is done from the rear. When the history exceeds a certain length, the oldest record is dropped from the front.

7.5.2 A deque in a circular array

A deque is built from the circular queue with a counter by adding a little. The structure is the same: `typedef struct { int data[CAP]; int front, count; } Deque;`. `front` points to the first item, and `count` is the item count. Insertion at the rear and removal from the front are the same as the `enqueue` and `dequeue` functions of the queue with a counter. The two new operations are insertion at the front and removal from the rear:

```
bool push_front(Deque *d, int x) {
  if (d->count == CAP) return false;
  d->front = (d->front - 1 + CAP) % CAP;    /* one back */
  d->data[d->front] = x;  d->count++;
  return true;
}
bool pop_back(Deque *d, int *x) {
  if (d->count == 0) return false;
  *x = d->data[(d->front + d->count - 1) % CAP];
  d->count--;  return true;
}
```

`push_front` puts the new item one cell behind the current first item. To do so, `front` is moved back by one. To avoid the negative-remainder trap of modular arithmetic, the step backward is computed as `(front - 1 + CAP) % CAP`. For `front = 0` this expression gives `CAP - 1`: the cell behind 0 is the last cell of the array. `pop_back` finds the position of the last item with `(front + count - 1) % CAP`, reads the value in that cell, and only decreases the counter. When the last item is removed, `front` does not change, because the first item stays where it is. For completeness, the other two operations are given below:

```
bool push_back(Deque *d, int x) {
  if (d->count == CAP) return false;
  d->data[(d->front + d->count) % CAP] = x;   /* just after the last item */
  d->count++;
  return true;
}
bool pop_front(Deque *d, int *x) {
  if (d->count == 0) return false;
  *x = d->data[d->front];
  d->front = (d->front + 1) % CAP;            /* one forward */
  d->count--;
  return true;
}
```

None of the four functions contains a loop; each performs a few arithmetic operations and one array access. All of them cost *O(1)*. The memory cost is `CAP` items.

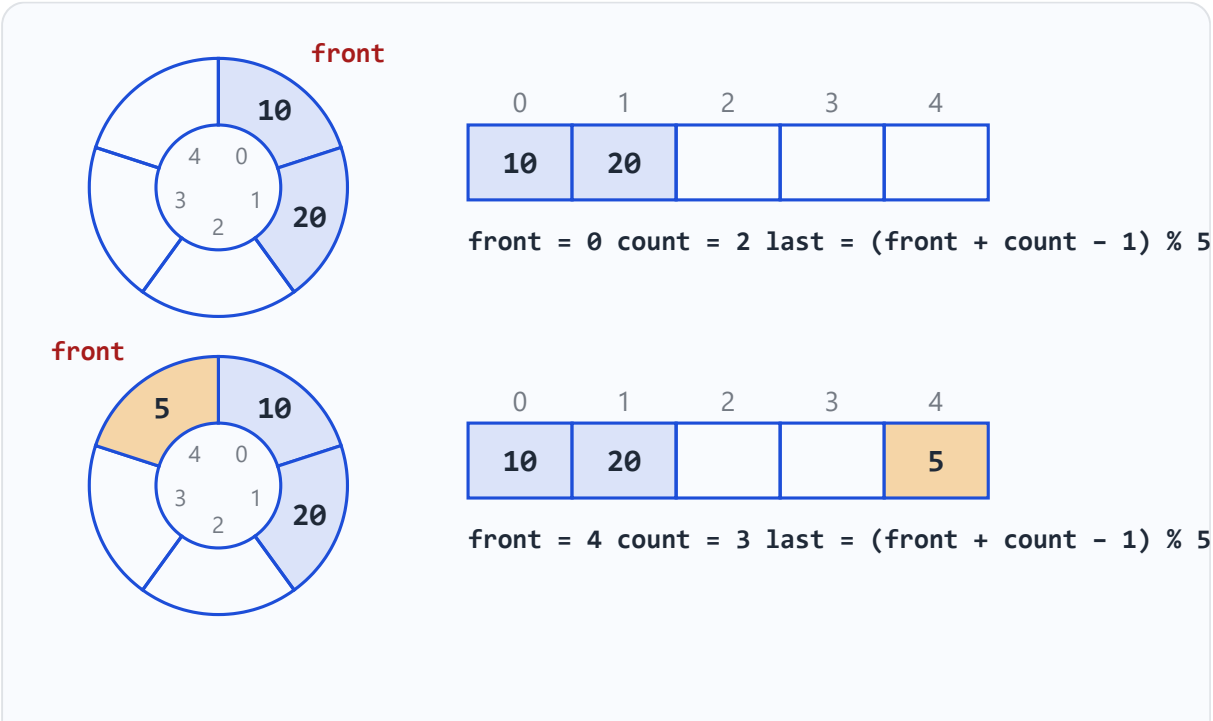
Example 7.4

Let `CAP = 5`. The values 10 and 20 have been inserted with `push_back`; `front = 0`, `count = 2`. The calls `push_front(&d, 5)`, `push_front(&d, 3)`, and two calls of `pop_back` are made in turn. Table 7.10 gives every step.

Call	Computation	`front`	`count`	`data[0..4]`	Order
Start	—	0	2	`10 20 . . .`	10 20
`push_front(&d, 5)`	`front` = (0 − 1 + 5) % 5 = 4	4	3	`10 20 . . 5`	5 10 20
`push_front(&d, 3)`	`front` = (4 − 1 + 5) % 5 = 3	3	4	`10 20 . 3 5`	3 5 10 20
`pop_back`	pos = (3 + 4 − 1) % 5 = 1; `*x` = 20	3	3	`10 . . 3 5`	3 5 10
`pop_back`	pos = (3 + 3 − 1) % 5 = 0; `*x` = 10	3	2	`. . . 3 5`	3 5

Table 7.10: Trace of a deque in a circular array (`CAP = 5`). The last column is the logical order of the deque from front to back.

The first `push_front` shows that wrap-around works in the backward direction as well. `front` goes from 0 to 4 and 5 is written into the last cell of the array. In the logical order, however, 5 is in front of 10. Read flat, the array shows 10, 20, ·, ·, 5; the order of the deque is read starting at `front` and wrapping around: 5, 10, 20. The two calls of `pop_back` return not the most recently inserted items but the items at the *end* of the deque: first 20, then 10. If a `pop_front` were called at this point, it would return 3 and `front` would advance to 4.



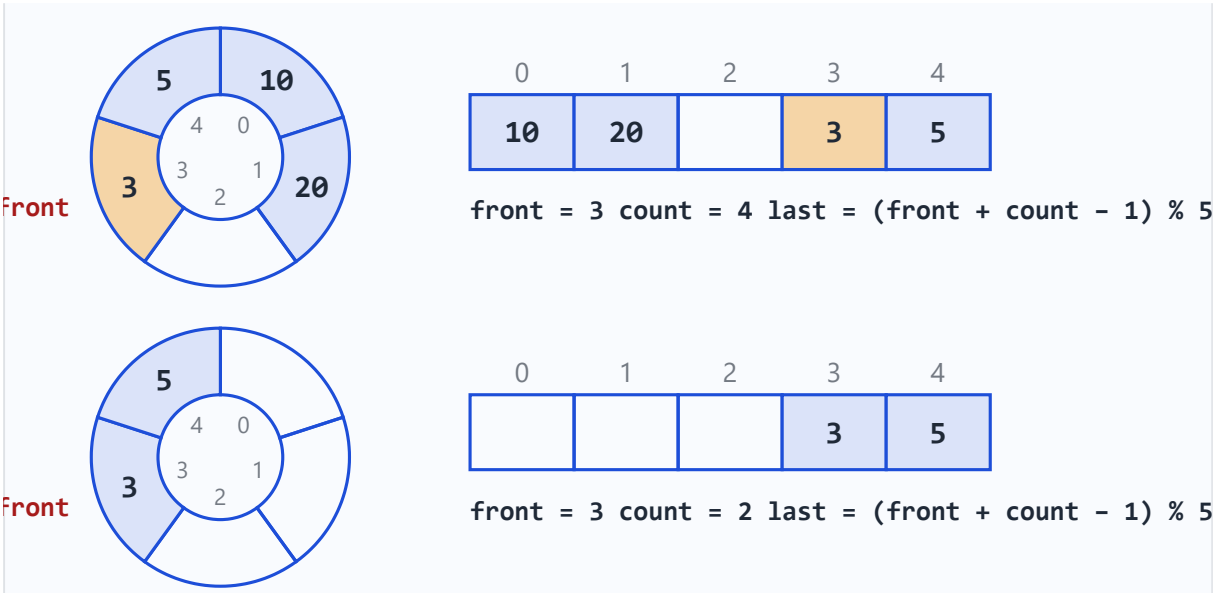


Figure 7.12: Four frames from the deque trace. From top to bottom: the start (10, 20); 5 pushed at the front, with `front` wrapping from 0 to the end of the array (4); 3 pushed at the front as well (`front = 3`); after two `pop_back` calls only 3 and 5 remain. The bottom line of each frame computes the position of the last item.

Common mistake

Writing the step backward as `(front - 1) % CAP` . In C the value of `(0 - 1) % 5` is −1, not 4 as in mathematics. The code writes to `data[-1]` . This address falls on the memory just before the array, often another field of the structure or another variable. The compiler does not catch it. In backward steps, `+ CAP` must always be added.

7.5.3 Several queues in one array

Some programs use many queues at the same time. For example, an operating system may keep a separate ready-process queue for each priority level. Reserving a separate large array for each queue wastes memory, because not all the queues fill up at once. A method used for stacks can be tried here too: a one-dimensional array `memory[]` is divided into *n* segments and each queue is given one segment. For the *i*-th queue three values are kept: `front[i]` , `rear[i]` , and `boundary[i]` , which marks where the segment ends.

Two stacks fit into one array very efficiently: one grows from the beginning of the array to the right, the other from the end to the left, and they meet only when the whole array is full. This trick of growing from the ends does not work for queues. The bottom end of a stack is fixed, but both ends of a queue move. A queue keeps drifting within its segment as items are inserted and removed. If each segment is managed as a circular queue, no shifting is needed. But when a segment fills up, even if other segments have free space, one of the queues must be moved to make room. This takes *O(m)* time, where *m* is the length of the array. Because of this complexity, nowadays each queue usually gets its own growing array or linked list when several queues are needed.

7.5.4 Round-robin job scheduling

On a computer with a single processor, many processes want to run at the same time. The operating system shares the processor among the processes. The best-known sharing method is **time-sliced scheduling**, also called **round robin**. Ready processes wait in a queue. The scheduler takes the process at the front and runs it for at most *Q* units, one time slice. If the process finishes its work in this time, it leaves the system. If it cannot finish, it returns to the end of the queue with its remaining work. Thus no process can monopolize the processor, and every process gets its turn at regular intervals.

In the simulation the processes are represented by the numbers 0, 1, 2, and the amounts of remaining work by the array `left[]`. The queue is the circular queue with a counter; the function names are `enqueue` and `dequeue` again:

```
#define CAP 16
typedef struct { int data[CAP]; int front, count; } Queue;

bool enqueue(Queue *q, int x) {
  if (q->count == CAP) return false;
  q->data[(q->front + q->count) % CAP] = x;
  q->count++;
  return true;
}
bool dequeue(Queue *q, int *x) {
  if (q->count == 0) return false;
  *x = q->data[q->front];
  q->front = (q->front + 1) % CAP;
  q->count--;
  return true;
}
```

Initially all three processes are in the queue. P0 has 5 units of work, P1 has 2, and P2 has 4, and the time slice is *Q = 2*:

```
int left[] = {5, 2, 4};      /* work left for P0, P1, P2 */
int Q = 2, time = 0, p;
Queue q = { {0}, 0, 0 };      /* empty queue with counter */
for (int i = 0; i < 3; i++)
  enqueue(&q, i);             /* all start in the queue */
```

The scheduler itself is a single loop:

```
/* q: queue of process ids; Q: time slice */
while (dequeue(&q, &p)) {
  int run = left[p] < Q ? left[p] : Q;
  time += run;
  left[p] -= run;
  if (left[p] > 0)
    enqueue(&q, p);              /* back of line */
  else
    printf("P%d done: t=%d\n", p, time);
}
```

The loop runs until the queue is empty; because `dequeue` returns `false` on an empty queue, the condition is also the termination check. In each round the process at the front is taken into `p`. The time the process will run in this round is the smaller of its remaining work and the time slice; the conditional expression (`? :`) computes this in one line. The clock (`time`) advances by that much, and the process's remaining work decreases by the same amount. If work remains, the process is added at the end of the queue; otherwise its finishing time is printed.

Round	Dequeued	`run`	Running interval	`left` (P0, P1, P2)	Result	Queue (front to rear)
—	—	—	—	5, 2, 4	—	P0 P1 P2
1	P0	2	0–2	3, 2, 4	goes back to the rear	P1 P2 P0
2	P1	2	2–4	3, 0, 4	**P1 done: t=4**	P2 P0
3	P2	2	4–6	3, 0, 2	goes back to the rear	P0 P2
4	P0	2	6–8	1, 0, 2	goes back to the rear	P2 P0
5	P2	2	8–10	1, 0, 0	**P2 done: t=10**	P0
6	P0	1	10–11	0, 0, 0	**P0 done: t=11**	(empty)

Table 7.11: Trace of round robin with P0 = 5, P1 = 2, P2 = 4 units of work and $Q = 2$.

The program prints `P1 done: t=4`, `P2 done: t=10`, and `P0 done: t=11` in this order. In the last round P0 has only 1 unit of work left, so the slice is 1, not 2. The total time equals the sum of the jobs (5 + 2 + 4 = 11), because the processor is never idle. The result of scheduling is the finishing order. The shortest job (P1) finishes first and the longest (P0) last. Had the processes instead been run without interruption under a first-come-first-served rule, the finishing times would have been 5 for P0, 7 for P1, and 11 for P2. A short job would have had to wait behind a long one.



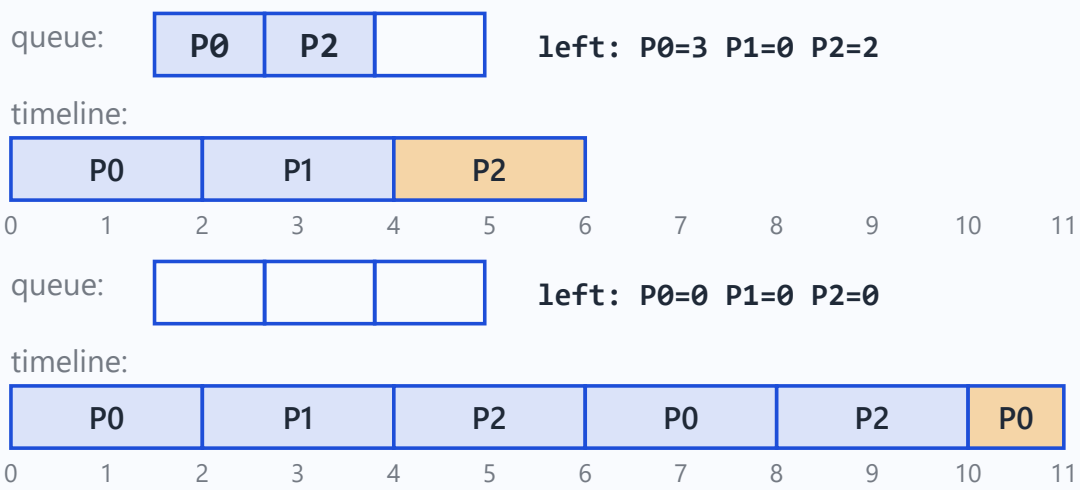


Figure 7.13: Three frames from the round-robin simulation. Each frame shows the queue and the remaining work at the top and the timeline at the bottom. From top to bottom: P0 has returned to the rear after its first slice; P2 has returned to the rear after its first slice ($t = 6$); all processes have finished ($t = 11$).

In terms of cost, each round performs one `dequeue` and at most one `enqueue`; both are $O(1)$. If the total work of a process is w units, the process is taken from the queue $\lceil w/Q \rceil$ times. The total cost of the simulation is proportional to the sum of these numbers. In the example this sum is 3 + 1 + 2 = 6 rounds. As the time slice gets smaller, processes get their turn more often, but the number of rounds increases. With $Q = 1$ the same example takes 11 rounds. In real systems every round carries the cost of a context switch, so the slice is not chosen very small.

7.5.5 A queue from two stacks

If a queue is needed in an environment that offers only stack operations, two stacks suffice. The idea rests on the fact that a stack reverses order. If a sequence of items is put onto one stack and transferred from there to another stack, it is reversed twice and thus returns to its original order. The queue consists of two stacks: `in` and `out`. The stack type `Stack` is defined by a `data` array and a `top` index that is −1 when the stack is empty. `push`, `pop`, and `stack_empty` are the familiar stack operations; `pop` returns `false` if the stack is empty. The queue type has the form `typedef struct { Stack in, out; } Queue2;`.

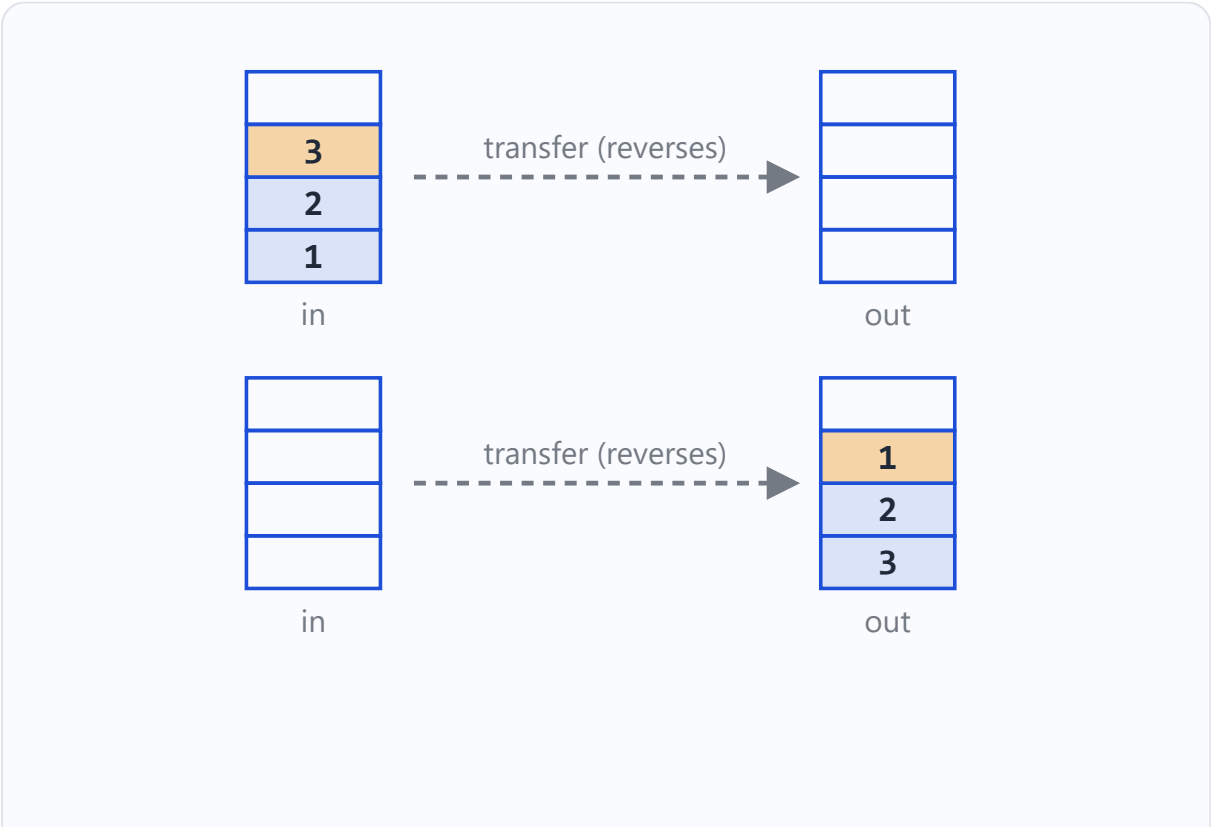
```
void enqueue(Queue2 *q, int x) {
  push(&q->in, x);
}
bool dequeue(Queue2 *q, int *x) {
  int t;
  if (stack_empty(&q->out))      /* out is empty */
    while (pop(&q->in, &t))      /* move them all */
      push(&q->out, t);
  return pop(&q->out, x);
}
```

`enqueue` always puts the item on the in-stack; its cost is a single `push`. If the out-stack is empty, `dequeue` first transfers all the items of the in-stack to the out-stack one by one. The transfer reverses the order: the item at the very bottom of the in-stack, that is, the oldest item, ends up on top of the out-stack. Then one `pop` is made from the out-stack. If the out-stack is not empty, no transfer is made, because every item in the out-stack is older than every item in the in-stack. If the transfer were made before the out-stack was empty, the new items would land on top of the old ones and the order would break. If both stacks are empty, the last `pop` returns `false` and `dequeue` returns `false` too. `enqueue` ignores the return value of `push`. If the in-stack becomes full, the item is silently lost; a safe version must pass this value on to the caller.

Operation	What happens?	`in`	`out`	Returned
enqueue 1, 2, 3	three `push` calls	1 2 3	—	—
dequeue	out is empty: 3, 2, 1 are transferred; then `pop`	—	3 2	1
enqueue 4	`push`	4	3 2	—
dequeue	out is not empty: only `pop`	4	3	2
dequeue	out is not empty: only `pop`	4	—	3
dequeue	out is empty: 4 is transferred; then `pop`	—	—	4

Table 7.12: Trace of a queue made of two stacks. Stacks are written from bottom to top.

The removal order is 1, 2, 3, 4, the same as the insertion order. Although 4 was added after 2 and 3, it does not come out before them. When 4 is added, 2 and 3 are already in the out-stack, and 4 is transferred only after the out-stack has emptied.



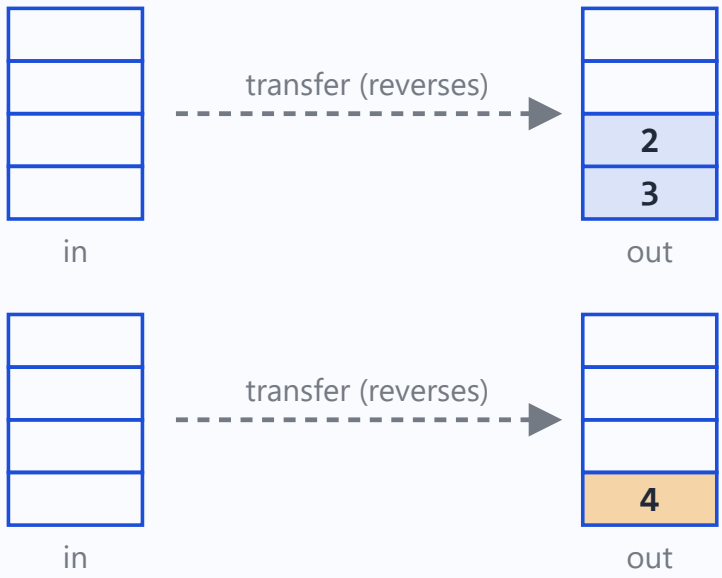


Figure 7.14: A queue made of two stacks. From top to bottom: 1, 2, 3 are in the in-stack; all three have been transferred to the out-stack and the oldest item, 1, is on top; 1 has been removed, leaving 2 and 3 in the out-stack; finally, when the out-stack has emptied, 4 has been transferred too and is the only item left to remove.

A single call of `dequeue` can be expensive: if the out-stack is empty and the in-stack holds n items, n transfers are made and the call takes $\Theta(n)$ time. But such expensive calls are rare. Over a whole sequence of operations, the cost per operation is constant.

Theorem 7.2 (Amortized cost of a queue made of two stacks)

Every sequence of m calls to `enqueue` or `dequeue`, starting from an empty queue, performs at most $4m$ stack operations (`push` or `pop`). Hence the amortized cost of both operations is $O(1)$.

Proof. Follow the life of a single item in the queue. The item is pushed onto the in-stack once by `enqueue` (one `push`). During a transfer it leaves the in-stack once and is put onto the out-stack once (one `pop`, one `push`). It never goes back from the out-stack to the in-stack, so it is never transferred a second time. Finally it leaves the out-stack once by `dequeue` (one `pop`). Each item thus takes part in at most four stack operations. In addition to these, each `dequeue` call performs at most one `pop` that fails when the stacks are empty and one `stack_empty` check. Since at most m items can be inserted in the sequence, the operations belonging to items number at most $4m$; the failing operations are also bounded by a constant factor. The total work is $O(m)$ and the average work per operation is $O(1)$. ■

The method in this proof is the simplest form of **amortized analysis**. The cost of an expensive operation is distributed over the cheap operations that make it possible. Here every `enqueue` "pays in advance" for the cost of later transferring its item. The theorem was also tested by an experiment.

In a randomly chosen sequence of 60 operations, 37 insertions were made and all the inserted items were eventually removed. A total of 74 `push` and 74 `pop` calls were counted; this means exactly two `push` and two `pop` calls per item. All items came out in insertion order. One point deserves attention here: amortized *O(1)* does not say that every call is fast. If an application requires that no operation exceed a certain time (for example, a real-time audio buffer), the circular queue is more suitable.

7.5.6 Stack or queue?

To decide whether a problem needs a stack or a queue, ask a single question: of the items waiting to be processed, which should be handled first, the newest or the oldest? If the newest, a stack is appropriate; if the oldest, a queue. If neither the newest nor the oldest but the item that is most important by some other criterion is to be taken first, a priority queue is needed. Table 7.13 classifies the examples in the slides according to this question.

Problem	Structure	Reason
Documents sent to a printer	Queue	Documents are printed in the order they were sent.
Ctrl+Z in a text editor	Stack	The most recent operation is undone first.
Printing a tree level by level	Queue	Level-order traversal (BFS): the node discovered first is processed first.
Checking nested parentheses	Stack	A closing parenthesis matches the most recently opened one.
Function calls	Stack	The call stack: the most recently called function returns first.
Most seriously ill patient first in an emergency room	Priority queue	The turn depends on priority, not on arrival; it is not FIFO.

Table 7.13: Suitable structures for problems.

The last row can be misclassified by looking at the word "queue" in its name. The instructor's note in the slides also points out this confusion. In the queue of everyday language, items are added at the end and taken from the front. A priority queue does not obey this rule; the word in its name carries only the meaning "a place to wait".

7.6 Complexity summary and measurements

Table 7.14 brings together the operation costs of the four implementations examined in these notes. The justification of each value in the table is given in the relevant section. Here only the common points are summarized.

Operation	Linear array	Circular array	Linked	Deque (circular)
Enqueue	*O(1)**	*O(1)*	*O(1)*	*O(1)*, at both ends
Dequeue	*O(1)*	*O(1)*	*O(1)*	*O(1)*, at both ends

Operation	Linear array	Circular array	Linked	Deque (circular)
Front / empty / full	*O(1)*	*O(1)*	*O(1)*	*O(1)*
Item count	`rear - front`	`(rear-front+MAX)%MAX`	*O(n)* or a counter	`count`
Memory	`MAX`	`MAX`	n · (item + pointer)	`CAP`

Table 7.14: Complexity summary of queue implementations. n is the number of items.

The starred value carries a caveat. In a linear queue, insertion by itself is *O(1)*, but if false full is cured by shifting, insertion rises to *O(n)*. In a linked queue, if no separate counter is kept, the item count can be found only by walking the list from beginning to end. Adding a counter variable brings this operation down to *O(1)* as well. Because insertion and removal take constant time in all the implementations, the cost of an algorithm that uses a queue is determined by the number of queue operations it performs. The round-robin simulation and level-order traversal perform *O(n)* queue operations for n items.

The difference between the linear queue with shifting and the circular queue was also shown by measurement. Both queues used an array of M cells. The shifting queue held $M - 1$ items at all times, and the circular queue held $M - 2$ items, since one cell stays empty there. One insertion and one removal were then performed 200,000 times. The program was compiled with `-O2` and the times were measured with `timespec_get`. The results are given in Table 7.15. Absolute times vary from machine to machine; what matters is how the columns change with M.

M	Shifting: items moved	Shifting: time	Circular: time
1000	199.799.001	7–8 ms	≈ 1 ms
2000	399.798.001	13–15 ms	≈ 1 ms
4000	799.796.001	31–33 ms	≈ 1 ms
8000	1.599.792.001	63–76 ms	≈ 1 ms

Table 7.15: 200,000 insertion–removal pairs on a nearly full queue. The times are the range of two runs and vary from machine to machine.

The number of items moved doubles when M doubles; each insertion moves about $M - 1$ items. The times grow at the same rate. The time of the circular queue, on the other hand, is independent of M. The times of the shifting version are surprisingly short relative to the number of items moved. The reason is that the compiler turns the shifting loop into a fast operation that copies blocks of memory in bulk. Even so, the growth is linear, and the gap widens as the queue gets larger.

7.7 Common mistakes

The six most common mistakes in queue code are collected below. Each was treated in detail in the earlier sections; here the symptom and the remedy are given together.

1. **Mistaking `rear == MAX - 1` for true fullness in a linear queue.** The condition says only that no room is left at the rear; the item count is `rear - front` and may be very small. The remedy is the circular queue.
2. **Forgetting `% MAX` in a circular queue.** At the first wrap-around the index goes outside the array. The program may run correctly for a long time and then break in an unexpected place. The remainder operation must be looked for on every line that changes an index.
3. **Advancing `rear` first and not restoring it if the queue is full.** In a version that returns an error, an attempted insertion into a full queue makes `rear` equal to `front` and the queue looks empty. The new position must be tried first in a temporary variable.
4. **Writing `(i - 1) % MAX` for a step backward.** In C the remainder of a negative number is negative; for `i = 0` the result is −1. The correct form is `(i - 1 + MAX) % MAX`.
5. **Not setting `rear = NULL` when the last item is removed from a linked queue.** `rear` points to the freed node; the next insertion writes into that memory and the new item is not visible in the queue.
6. **Forgetting to update `front` at the first insertion into an empty linked queue.** Either the write goes through a `NULL` pointer and the program crashes, or `front` stays `NULL` and the inserted items can never be removed.

The common root of these mistakes is edge cases: the empty queue, the full queue, the one-item queue, and the last cell of the array. When a queue implementation is tested, at least the following scenarios should be tried: removing from an empty queue, filling the queue completely and trying one more insertion, at least one wrap-around, and emptying the queue entirely and reusing it.

7.8 Exercises and solutions

This section gives the five exercises in the slides with detailed solutions. The solutions that contain code have been compiled and tested.

7.8.1 Item count and positions

Question. How many items does a circular queue with `MAX = 6`, `front = 4`, and `rear = 1` hold, and at which indices are they?

Solution. The item count is `(rear - front + MAX) % MAX = (1 - 4 + 6) % 6 = 3`. Because `front` points to the cell before the first item, the items begin at the cell after `front`: `(4 + 1) % 6 = 5`. Then, with wrap-around, come 0 and 1. The items are in cells 5, 0, and 1, and the front item is in cell 5. Cells 2, 3, and 4 are empty; cell 4 is the cell that the first method leaves empty.

7.8.2 Circular queue with a flag

Question. Write the insertion and removal routines of a circular queue that tells full from empty with a flag.

Solution. A boolean variable called `last_was_enqueue` records whether the last operation was an insertion. When `front == rear`, the queue is full if this variable is true and empty if it is false. Initially the queue is empty, so its value is `false`.

```
#define MAX 4
int queue[MAX];
int front = 0, rear = 0;
bool last_was_enqueue = false;      /* was the last op an enqueue? */

bool enqueue(int x) {
    if (front == rear && last_was_enqueue) return false;   /* full */
    rear = (rear + 1) % MAX;
    queue[rear] = x;
    last_was_enqueue = true;
    return true;
}
bool dequeue(int *x) {
    if (front == rear && !last_was_enqueue) return false;  /* empty */
    front = (front + 1) % MAX;
    *x = queue[front];
    last_was_enqueue = false;
    return true;
}
```

After the full check, insertion works in the same way as the code in the slides, but at the end it raises the flag. Removal lowers the flag. The flag is read only when `front == rear`. While the queue holds between 1 and `MAX - 1` items, the two indices differ anyway and the value of the flag does not matter. When this version was tested, four insertions fit into the empty four-cell queue (`front = rear = 0`, flag true) and the fifth was rejected. After one removal (10) and one insertion (99), the queue was full again and the items came out in the order 11, 12, 13, 99. No cell is wasted; the cost is one assignment in every operation and one boolean comparison in the full or empty check.

7.8.3 Cost of a queue made of two stacks

Question. Write a queue using two stacks. What is the amortized cost of `dequeue`?

Solution. The code consists of the `enqueue` and `dequeue` functions in the slides: insertion does a `push` onto the in-stack; removal transfers the whole in-stack if the out-stack is empty and then does a `pop` from the out-stack. As shown in the proof of Theorem 7.2, each item is pushed onto a stack at most twice and popped from a stack twice. The total cost of a sequence of m operations is $O(m)$. The amortized cost of `dequeue` is therefore $O(1)$. A single call in the worst case, however, takes $\Theta(n)$. For

example, add 1, 2, 3, 4, remove one item, add 5, and remove four more. The output is 1, 2, 3, 4, 5. The first removal transfers four items and the last removal transfers only 5; the total number of `push` calls is 10, made up of 5 insertions and 5 transfers.

7.8.4 Circular linked queue with only a rear pointer

Question. Write a queue using a circular linked list in which only the `rear` pointer is kept.

Solution. In a circular linked list the link of the last node is not `NULL` but the first node. A separate `front` pointer is therefore unnecessary: the front node is always `rear->link`. In a one-node queue the node points to itself. In an empty queue `rear` is `NULL`.

```
bool enqueue(node **rear, int x) {
  node *new_node = malloc(sizeof *new_node);
  if (new_node == NULL) return false;
  new_node->data = x;
  if (*rear == NULL) new_node->link = new_node;  /* lone node points to itself */
  else {
    new_node->link = (*rear)->link;         /* new node points to front */
    (*rear)->link = new_node;               /* old rear points to new */
  }
  *rear = new_node;
  return true;
}
bool dequeue(node **rear, int *x) {
  if (*rear == NULL) return false;          /* empty */
  node *front = (*rear)->link;              /* front follows rear */
  *x = front->data;
  if (front == *rear) *rear = NULL;          /* last node gone */
  else (*rear)->link = front->link;
  free(front);
  return true;
}
```

Insertion places the new node between the last node and the front node. First the new node points to the front node (`new_node->link = rear->link`), then the old last node points to the new node, and finally `rear` moves to the new node. The order of these links matters: if the second assignment were made first, the address of the front node would be lost. Removal takes out the front node (`rear->link`). If the queue has a single node, front and rear are the same node and the queue becomes empty. Otherwise the link of the last node is turned to the second node. Both operations are *O(1)* and only one pointer is kept. In testing, when 10, 20, and 30 were added and removed, the order was preserved; all 1003 `malloc` calls were matched by `free`.

7.8.5 Round robin, *Q = 3*

Question. Run the round-robin example (P0 = 5, P1 = 2, P2 = 4) with *Q = 3*. When do the processes finish?

Solution. P0 runs from 0 to 3 and has 2 units left; it returns to the rear. P1 runs from 3 to 5 and finishes at **t = 5**. P2 runs from 5 to 8 and has 1 unit left; it returns to the rear. P0 runs its remaining 2 units from 8 to 10 and finishes at **t = 10**. P2 runs its last unit from 10 to 11 and finishes at **t = 11**. The program prints `P1 done: t=5`, `P0 done: t=10`, `P2 done: t=11`. The finishing order has changed relative to *Q = 2*: P0 now finishes before P2. The number of rounds also falls from 6 to 5. As the slice grows, scheduling approaches first-come-first-served; for *Q ≥ 5* the finishing times are 5, 7, and 11.

7.9 Summary

- A **queue** is an ordered list in which insertion takes place at the rear and removal at the front; its rule is **first in, first out** (FIFO). A stack reverses order and a queue preserves it. The basic operations are `enqueue` (overflow), `dequeue` (underflow), `queue_front`, `queue_empty`, and `queue_full`.
- The **linear queue** uses two indices: `front` points to the cell before the first item and `rear` to the last item; the empty condition is `front == rear`. Because the indices only move forward, cells vacated at the front cannot be used (**false full**). Shifting solves this problem but makes insertion *O(n)*.
- The **circular queue** wraps the indices with `% MAX`. Empty: `front == rear`; full: `(rear + 1) % MAX == front`. The item count is `(rear - front + MAX) % MAX` and a step backward is `(i - 1 + MAX) % MAX`; in C the remainder of a negative number is negative.
- Two indices cannot distinguish `MAX + 1` occupancy states. The remedy is either to leave one cell empty (capacity `MAX - 1`) or to keep a **counter** or a **flag**. The counter version gives the item count in *O(1)* and computes `rear`.
- An operation that can fail must not modify the structure when it fails: the new position is first tried in a temporary variable.
- The **linked queue** inserts at the tail and deletes from the head with `front` and `rear` pointers; both operations are *O(1)*. At the first insertion into an empty queue `front` is updated, and when the last item is removed `rear = NULL`; otherwise a leak or a dangling pointer results.
- In practice, a circular array is used for fixed-capacity buffers and a circular array that doubles in size (amortized *O(1)*) for queues with an unknown upper bound; when growing, the items are copied in order starting at `front`.
- The **deque** inserts and removes at both ends; in a circular array all four operations are *O(1)*. It covers both the stack and the queue.
- In **round-robin** scheduling, processes wait in a queue, run for at most one time slice, and return to the rear if work remains. In a **queue made of two stacks**, each item is transferred at most once; the operations are amortized *O(1)*.

7.10 Key terms

queue FIFO (first in, first out) LIFO (last in, first out) front rear enqueue dequeue abstract data type overflow underflow circular queue modular arithmetic buffer breadth-first search (BFS) level-order priority queue dangling pointer memory leak deque (double-ended queue) amortized analysis doubling stack

7.11 Test Yourself

1. In a safe circular queue, `MAX_QUEUE_SIZE = 4` and initially `front = rear = 0`. The values 1, 2, 3, 4 are to be added in turn, one item is removed, and 4 is added again. Which insertion is rejected? At the end, what are `front`, `rear`, and the item count? Which value does the next `dequeue` return?

   Answer

   1, 2, 3 go into cells 1, 2, 3 respectively. For 4, `next = (3 + 1) % 4 = 0 = front`, and the insertion is rejected. The removal advances `front` to 1 and returns 1. The re-added 4 is written into cell `rear = 0`. At the end `front = 1`, `rear = 0`, and the item count is `(0 − 1 + 4) % 4 = 3`. The next removal returns 2.

2. Three items are added to a linear queue with `MAX_QUEUE_SIZE = 5` and all three are removed. Although the queue is empty, how many more insertions can be made at most? Why?

   Answer

   After three insertions and three removals, `front = rear = 2`. Insertion is possible until `rear` reaches 4: two insertions. At the third attempt `rear ≥ 4` holds and the queue counts as full. The cause is false full: cells 0, 1, and 2 are empty, but because the indices only move forward they are never used again.

3. A programmer has written the step backward in a deque as `d->front = (d->front - 1) % CAP;`. What happens if `push_front` is called when `CAP = 8` and `front = 0`?

   Answer

   In C the value of `(0 − 1) % 8` is −1. `front` becomes −1 and the item is written into `data[-1]`, that is, outside the array. This is undefined behavior; another field of the structure or a neighboring variable may be corrupted. The correct expression is `(front − 1 + CAP) % CAP`, which gives 7.

4. The line `if (*front == NULL) *rear = NULL;` has been deleted from the `dequeue` function of the linked queue. 5 and 7 are added to an empty queue, two items are removed, and then 9 is added. What happens?

   Answer

   After the second removal `front` is `NULL`, but `rear` still points to the freed node of 7. When 9 is added, `*rear != NULL`, so the code does not regard the queue as empty and writes into freed memory with `(*rear)->link = new_node`; this is undefined behavior. Because `front` is not updated, it stays `NULL`. The next `dequeue` finds the queue empty; 9 can never be removed and its node leaks.

5. In a circular queue with a counter, `CAP = 4`, `front = 3`, and `count = 3`. In which cells are the items? Into which cell will the next `enqueue` write its item?

   Answer

   Starting at `front`, the items are in cells 3, 0, and 1. The position of the last item is `(3 + 3 - 1) % 4 = 1`. The next insertion writes into cell `(3 + 3) % 4 = 2` and the counter becomes 4; the queue is full.

6. To an empty deque with `CAP = 6` (`front = 0`, `count = 0`), `push_front 1`, `push_front 2`, and `push_back 3` are applied in turn. Which value is in which cell? What does `pop_front` return?

   Answer

   The first `push_front` moves `front` to `(0 - 1 + 6) % 6 = 5` and writes 1 into cell 5. The second moves `front` to 4 and writes 2 into cell 4. `push_back 3` writes into cell `(4 + 2) % 6 = 0`. The order of the deque is 2, 1, 3; `pop_front` returns 2.

7. In a round-robin simulation each of three processes has 3 units of work and $Q = 2$. When do the processes finish?

   Answer

   In the first round P0 runs from 0 to 2, P1 from 2 to 4, and P2 from 4 to 6; each has 1 unit left and returns to the rear. In the second round P0 runs from 6 to 7, P1 from 7 to 8, and P2 from 8 to 9. The finishing times are 7 for P0, 8 for P1, and 9 for P2.

8. 1, 2, 3, 4 are added to a queue made of two stacks; one item is removed; 5 is added; four more

items are removed. What is the removal order? How many `push` calls are made in total?

Answer

At the first removal the out-stack is empty; 4, 3, 2, 1 are transferred and 1 is removed. 5 is pushed onto the in-stack. The next three removals return 2, 3, 4 from the out-stack. At the last removal the out-stack is empty; 5 is transferred and removed. The removal order is 1, 2, 3, 4, 5. The number of `push` calls is 10, made up of 5 insertions and 5 transfers.

9. In the first method (leaving one cell empty), why does a queue of `MAX` cells hold at most `MAX - 1` items? What would happen if the queue were allowed to hold `MAX` items?

Answer

If the `MAX`-th item were added, `rear` would advance and become equal to `front`. But `front == rear` is the condition for an empty queue. The code would mistake a full queue for an empty one: `dequeue` would report underflow, while `enqueue` would overwrite items. The difference of the two indices takes only `MAX` values, whereas the item count can take `MAX + 1` values; one state has to be given up.

7.12 Further reading

- E. Horowitz, S. Sahni, and S. Anderson-Freed, *Fundamentals of Data Structures in C* (2nd ed.), Sections 3.3 and 4.3: the source of the array-based and linked queue implementations followed in these notes, of the circular queue, and of several queues in one array.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms* (3rd ed.), Section 10.1: gives the queue in pseudocode on a circular array, briefly and precisely; the deque and the queue from two stacks appear as exercises.
- R. Sedgewick, *Algorithms in C*, Parts 1–4, Chapter 4: treats abstract data types, the stack and queue interfaces, and their array and linked-list implementations comparatively.

7.13 References

1. E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2nd ed., Sections 3.3 and 4.3.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 3rd ed., MIT Press, Section 10.1 (queues, deque exercises).
3. R. Sedgewick, *Algorithms in C*, Parts 1–4, Chapter 4.

Stack and Queue Applications

A stack is a simple structure that returns the most recently pushed item first. This simplicity makes it indispensable in many corners of programming. These notes treat two classic applications of the stack in detail. The first is searching for a way out of a maze: every fork in the path is pushed onto the stack, and when a dead end is reached, the search returns to the last fork. The second is the evaluation of arithmetic expressions. A compiler first converts an expression such as `a / b - c + d * e - a * c` to postfix notation according to operator precedence, and then evaluates it in a single pass. Both jobs are done with a stack. The notes close with other applications, such as parenthesis checking, undo/redo, and non-recursive tree traversal, and with the difference a queue makes on the same problems. For every routine the C code, a line-by-line explanation, a code trace, and a complexity analysis are given.

LEARNING OUTCOMES

Coding backtracking
Search a maze with a stack, return from a dead end to the last fork, and follow this process step by step.

Constructing the worst case
Design the input on which an algorithm does the most work and justify its cost with that input.

Converting notations
Convert between infix, prefix, and postfix notation by hand and with a stack.

Evaluating a postfix expression
Run the `eval` routine step by step with an operand stack, and get the order in which operands are popped right.

Coding precedence
Write the routine that converts an infix expression to postfix using an `isp` / `icp` table, and make the right decision on equal precedence.

Generalizing the stack
Use the stack for parenthesis checking, for problems such as undo/redo, and for non-recursive traversal; use the queue for shortest-path search.

8.1 The stack and backtracking

All the routines in these notes are built on a single structure, so let us first recall the stack briefly. A **stack** is a linear structure in which items are inserted at one end only and removed from the same end. This end is called the **top**. Insertion is called "putting on top" (`push`) and removal "taking from the top" (`pop`). The item pushed last comes out first. This rule is called **last in, first out** (LIFO).

Definition 8.1

A **stack** is an ordered collection of items in which insertion and removal take place only at the end called the top. In a stack implemented with an array, the items sit in cells `stack[0]` , `stack[1]` , ...; the variable `top` holds the index of the topmost item. In an empty stack, `top == -1` . Pushing is done with `stack[++top] = x` and popping with `x = stack[top--]` ; both take *O(1)* time.

In the array implementation there are two boundary cases. When the array is full, no new item can be pushed; this is called **overflow**. No item can be popped from an empty stack; this is called **underflow**. The short code in these notes does not write these checks in most places. For example, the `eval` routine assumes that the stack never runs empty on a well-formed expression. The `push` and `pop` functions used in the verification programs, on the other hand, check both cases. In a real program these checks must not be skipped.

The "last in, first out" behavior of the stack fits one kind of problem naturally. In any problem that proceeds by making a series of decisions, where a wrong decision sometimes has to be undone, the most recent decision is the first to be undone. For example, while moving through a maze, a direction is chosen at every fork. If the chosen path ends in a dead end, one returns to the nearest fork, that is, the one passed most recently, and tries another direction there. This method is called **backtracking**.

Definition 8.2

Backtracking is a search method that extends a partial solution step by step and, when it reaches a dead end, undoes the last choice and tries the next option. The choices are kept on a stack: a new choice is pushed, and a return is made by popping the top choice.

Backtracking can also be written recursively. In that case the programmer does not build the stack; every recursive call adds a frame to the **call stack**, and the return of the call performs the step back. The maze routine, however, builds the stack explicitly, as an array. In this way what is on the stack can be seen and traced at every moment. This is the best way to understand the role of the stack in backtracking.

8.2 The maze problem

The maze problem is the classic example of backtracking with a stack. There is an entrance, an exit, and passages divided by walls between them. A path from the entrance to the exit is sought. The problem looks simple, but its details are instructive. How are the directions coded, how is entering the same cell twice prevented, and from where does the search continue after a return? The answers to these questions show how important the choice of data structure is.

8.2.1 Representing the maze with a matrix

The maze is represented by a two-dimensional integer matrix. Each cell of the matrix is a square of the maze. The value `0` stands for an open path and the value `1` for a wall. Figure 8.1 shows the maze used in these notes. Gray squares are walls and white squares are open paths. The entrance is cell `[1][1]`, marked with the letter S, and the exit is cell `[6][6]`, marked with the letter X. The size of the maze is written *m* × *p*: *m* rows and *p* columns. Here the interior is 6 × 6.

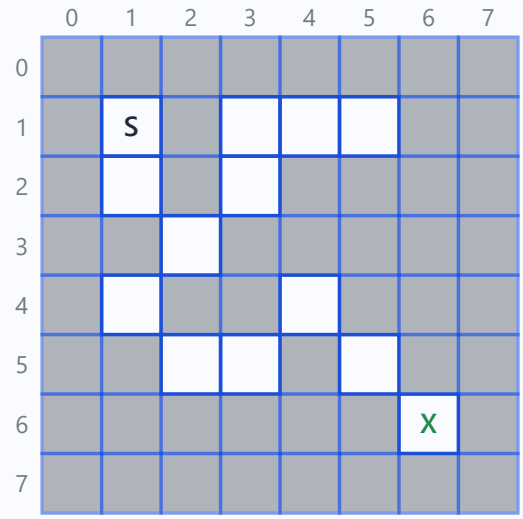


Figure 8.1: The maze matrix. Gray squares are walls (1) and white squares are open paths (0). Entrance S = `[1][1]`, exit X = `[6][6]`. Row 0, row 7, column 0, and column 7 on the outer frame are walls throughout.

The matrix in the figure is 8 × 8, although the maze itself is 6 × 6. The two extra rows and two extra columns are a deliberate trick. The border of the matrix is surrounded all the way around with `1`, that is, with wall. The reason is this. Looking north from a cell on the edge, for example from `[1][1]`, one goes to cell `[0][1]`. Without the frame this cell would lie outside the matrix: an invalid index such as `[-1][0]` would arise in a 6 × 6 array. C does not check array bounds. Reading outside an array is undefined behavior; the program either crashes or mistakes some random memory value for a wall or a path.

Without the frame, four conditions of the form "am I leaving the matrix?" would have to be checked at every step. The frame makes this check unnecessary. Every cell on the outer ring is a wall, and a wall is never entered. The algorithm therefore never steps out of the interior, and every neighbor index stays valid. Extra cells of this kind, which eliminate edge cases, are generally called **sentinels**. The cost is very small: a matrix of $(m + 2)(p + 2)$ cells.

Freedom of movement is also part of the problem. In this maze, in addition to horizontal and vertical moves, diagonal moves are also allowed. From a cell one can go to its eight neighbors. Diagonal moves make it possible to get around the tight corners of the maze. For example, one can go from `[2][1]` to `[3][2]` only with a diagonal step, because `[3][1]` and `[2][2]` in between are walls.

The algorithm has two more needs. First, the same cell must not be entered a second time. Otherwise one could go back and forth between two cells forever. The visited cells are therefore marked in a separate `mark` matrix. Second, when a dead end is reached, one must return to the last fork. The most recently passed fork is the first place to return to; this is exactly the definition of a stack.

8.2.2 Eight directions and the `move` table

The eight directions are numbered from 0 to 7. Direction 0 is north; the numbers increase clockwise. Direction 1 is northeast, 2 east, 3 southeast, 4 south, 5 southwest, 6 west, and 7 northwest. Figure 8.2 shows the eight neighbors of a cell and the index of each neighbor.



Figure 8.2: The eight directions of movement. The cell in the middle is `[i][j]`. Direction 0 is north (`[i-1][j]`); the following directions clockwise are 1…7.

Moving in a direction means adding a small number to the row index and to the column index. Going north decreases the row by one and leaves the column unchanged: $(-1, 0)$. Going southeast increases both by one: $(+1, +1)$. These eight pairs are collected in Table 8.1.

dir	Name	`move[dir].vert`	`move[dir].horiz`
0	N (north)	−1	0
1	NE (northeast)	−1	1
2	E (east)	0	1

dir	Name	`move[dir].vert`	`move[dir].horiz`
3	SE (southeast)	1	1
4	S (south)	1	0
5	SW (southwest)	1	−1
6	W (west)	0	−1
7	NW (northwest)	−1	−1

Table 8.1: The `move` array. The new position is `[i + vert][j + horiz]`.

This table is called a **look-up table**. It means reading a precomputed array instead of computing the value with conditions every time. Without the table, eight separate `if` statements would be written: "if the direction is 0, decrease the row by one; if it is 1, decrease the row and increase the column..." The table reduces these eight branches to a single line:

```
next_row = row + move[dir].vert;
next_col = col + move[dir].horiz;
```

In this way the direction becomes an ordinary loop variable. To try all eight directions it is enough to increase `dir` from 0 to 7. Changing the order in which directions are tried also becomes easy: only the order of the rows of the table changes, and the code stays the same.

8.2.3 Data definitions

All the constants, types, and global variables used by the routine are collected in the following declarations from the slides:

```
#define MAX_STACK_SIZE 100
#define EXIT_ROW 6
#define EXIT_COL 6
typedef struct { short int vert, horiz; } offsets;
offsets move[8] = {{-1, 0}, {-1, 1}, {0, 1}, {1, 1},
                   {1, 0}, {1, -1}, {0, -1}, {-1, -1}};
typedef struct { short int row, col, dir; } element;
element stack[MAX_STACK_SIZE];
int top = -1;
short int mark[8][8];
short int maze[8][8] = {{1,1,1,1,1,1,1,1},
    {1,0,1,0,0,0,1,1}, {1,0,1,0,1,1,1,1},
    {1,1,0,1,1,1,1,1}, {1,0,1,1,0,1,1,1},
    {1,1,0,0,1,0,1,1}, {1,1,1,1,1,1,0,1},
    {1,1,1,1,1,1,1,1}};
```

- `MAX_STACK_SIZE` is the size of the stack array. `EXIT_ROW` and `EXIT_COL` are the coordinates of the exit cell.
- The `offsets` structure holds the two components of a direction. The `move[8]` array is the table of Table 8.1 itself; the first pair, `{-1, 0}`, corresponds to direction 0 (north). Since the values are small, `short int` is sufficient.
- `element` is the record pushed onto the stack. It has three fields: the row of a cell, its column, and the direction from which the search should continue if we return to that cell (`dir`). This third field is very important. If only the position were kept on the stack, the directions of that cell would be tried again from 0 on a return. The direction field stores the information "in this cell I have looked as far as this direction".
- The `stack` array is the stack itself; `top = -1` says that the stack is empty.
- `mark` marks the visited cells. Because it is a global array, C fills it with zeros when the program starts; no initial value is needed. If it were a local array, its contents would be indeterminate.
- `maze` is an 8 × 8 matrix surrounded by 1s. Each inner pair of braces is a row; the first row is all 1s.

The maximum number of items the stack will hold can also be estimated in advance. A cell is pushed onto the stack only when moving from it to a new, unvisited cell. Since each open cell is visited at most once, the number of items on the stack cannot exceed the number of open cells. The interior of this maze has 13 open cells (the verification program also counted 13). The value 100 of `MAX_STACK_SIZE` is more than enough for this maze. For a general $m \times p$ maze, a safe upper bound is $m \cdot p$.

8.2.4 The `path` routine

The path-finding routine is called `path`. To keep the code readable, the slides give it in two parts; the two parts follow one another below:

```
void path(void) {
  short int row, col, next_row, next_col;
  int i, dir, found = FALSE;
  element position;
  stack[0].row = stack[0].col = 1;
  stack[0].dir = 1;     /* push the start */
  mark[1][1] = 1;  top = 0;
  while (top > -1 && !found) {
    position = pop(&top);     /* next one */
    row = position.row;
    col = position.col;
    dir = position.dir;       /* which way */
    while (dir < 8 && !found) {
      next_row = row + move[dir].vert;
      next_col = col + move[dir].horiz;
```

```
      if (next_row == EXIT_ROW &&
          next_col == EXIT_COL)
        found = TRUE;         /* the exit? */
      else if (!maze[next_row][next_col] &&
               !mark[next_row][next_col]) {
        mark[next_row][next_col] = 1;
        position.row = row;   /* came from */
        position.col = col;
        position.dir = ++dir; /* resume dir */
        push(&top, position);
        row = next_row;  col = next_col;
        dir = 0;
      } else ++dir;           /* try another */
    }                         /* back up */
  }
```

The code consists of two nested loops. The outer loop takes a position from the stack; the inner loop advances from that position as far as it can. Let us look at it line by line.

- At the start, the entrance cell `(1, 1)` is pushed onto the stack. Its direction field is 1. Direction 0 is north, and `[0][1]` is a wall on the frame; there is no need to try it. The entrance cell is marked in the `mark` matrix and `top = 0`.
- The outer loop runs as long as the stack is not empty and the exit has not been found. In each round a `position` is popped from the stack. `row`, `col`, and `dir` are read from this record. In other words: "I am in this cell and will start trying from this direction."
- The inner loop runs as long as `dir` has not reached 8 and the exit has not been found. In each round a neighbor is computed: `(next_row, next_col)`.
- If the neighbor is the exit cell, `found` becomes true and both loops end.
- If the neighbor is an open path (`!maze[…]`) and has not been visited before (`!mark[…]`), the search moves into that cell. First the neighbor is marked. Then the current cell is pushed onto the stack, but `++dir` is written into the direction field. This says: "if I return, let me continue from the next direction". Then `row` and `col` are turned to the new cell, and with `dir = 0` all directions of the new cell are tried from the start.
- If the neighbor is a wall or has been visited, only `++dir` is executed and the next direction is tried.
- If the inner loop ends with `dir == 8`, all eight directions of that cell are exhausted; the cell is a dead end. The code does nothing in this case. In the next round the outer loop pops the last record from the stack, and so a return to the previous cell takes place. The comment "back up" in the code refers to this moment.

A few details need attention. First, the exit check is done before the wall check. In this maze the exit cell is open and the order does not matter. But the exit cell is never marked in the `mark` matrix; the search ends the moment it is seen. Second, the record pushed onto the stack is not the new cell but the cell that was left. The new cell sits in the variables `row` and `col`. When the exit is found, the complete path therefore consists of three parts: the positions on the stack (from `stack[0]` to

`stack[top]`), the cell where the search currently stands, and the exit cell. As the slides note, the last part of the routine prints these three parts in order; if the exit was not found, the message "No exit path found!" is printed. In the verification program this last part was completed as follows:

```
if (found) {
  printf("row col\n");
  for (i = 0; i <= top; i++)        /* path on stack */
    printf("%2d%5d\n", stack[i].row, stack[i].col);
  printf("%2d%5d\n", row, col);          /* last position */
  printf("%2d%5d\n", EXIT_ROW, EXIT_COL);
} else printf("No exit path found!\n");
```

Third, `found`, `TRUE`, and `FALSE` are a habit from before C99's `bool` type: they are integers defined with `#define FALSE 0` and `#define TRUE 1`. `push` and `pop` are the two familiar functions of the stack implementation; they take `top` by address because they change it. Fourth, the line `position.dir = ++dir` both computes the value to be written into the record and changes the local `dir`. Since `dir = 0` is written right afterward, the local change has no effect; what matters is the value written into the record.

Common mistake

Forgetting to mark the new cell. If the line `mark[next_row][next_col] = 1` is not written, the algorithm moves from `(1, 1)` to `(2, 1)`, looks north from there, and returns to `(1, 1)`. It goes back and forth between two cells, adding a record to the stack at every crossing. In the end the stack overflows or the program stays in an infinite loop. The visited mark is the guarantee of termination.

The step-by-step trace slide of the slides writes the same logic more briefly, abbreviating the variable names of the inner loop (`nr`, `nc`) and building the new record in one line:

```
while (top > -1 && !found) {
  position = pop(&top);
  row = position.row;  col = position.col;
  dir = position.dir;
  while (dir < 8 && !found) {
    nr = row + move[dir].vert;
    nc = col + move[dir].horiz;
    if (nr == EXIT_ROW && nc == EXIT_COL)
      found = TRUE;
    else if (!maze[nr][nc] &&
             !mark[nr][nc]) {
      mark[nr][nc] = 1;
      position = (element){row, col, ++dir};
      push(&top, position);
      row = nr;  col = nc;  dir = 0;
    } else ++dir;
} }
```

The expression `(element){row, col, ++dir}` here is a **compound literal**, introduced in C99. It builds an unnamed temporary `element` object and fills its fields in order. It does in one line the work of three separate assignments. This version was also run separately in the verification program and produced the same stack movements.

8.2.5 Code trace: backtracking with a stack

Let us run the routine on the maze of Figure 8.1. At each step the contents of the stack are written as `row,col,dir` triples, from bottom to top. Table 8.2 gives the stack operations in order; the rows are identical to the output produced by the verification program.

Step	Event	Move	Stack (bottom to top)	`top`
0	Start	—	1,1,1	0
1	pop (1,1), start from dir 1	—	(empty)	−1
2	push (1,1, dir 5)	(1,1) dir 4 (S) → (2,1)	1,1,5	0
3	push (2,1, dir 4)	(2,1) dir 3 (SE) → (3,2)	1,1,5 · 2,1,4	1
4	push (3,2, dir 2)	(3,2) dir 1 (NE) → (2,3)	… · 3,2,2	2
5	push (2,3, dir 1)	(2,3) dir 0 (N) → (1,3)	… · 3,2,2 · 2,3,1	3
6	push (1,3, dir 3)	(1,3) dir 2 (E) → (1,4)	… · 2,3,1 · 1,3,3	4
7	push (1,4, dir 3)	(1,4) dir 2 (E) → (1,5)	… · 1,3,3 · 1,4,3	5
8	(1,5) dead end; pop (1,4)	dir 3…7 blocked	… · 2,3,1 · 1,3,3	4
9	pop (1,3)	dir 3…7 blocked	… · 3,2,2 · 2,3,1	3
10	pop (2,3)	dir 1…7 blocked	1,1,5 · 2,1,4 · 3,2,2	2
11	pop (3,2)	continue from dir 2	1,1,5 · 2,1,4	1
12	push (3,2, dir 6)	(3,2) dir 5 (SW) → (4,1)	1,1,5 · 2,1,4 · 3,2,6	2

Step	Event	Move	Stack (bottom to top)	`top`
13	push (4,1, dir 4)	(4,1) dir 3 (SE) → (5,2)	… · 3,2,6 · 4,1,4	3
14	push (5,2, dir 3)	(5,2) dir 2 (E) → (5,3)	… · 4,1,4 · 5,2,3	4
15	push (5,3, dir 2)	(5,3) dir 1 (NE) → (4,4)	… · 5,2,3 · 5,3,2	5
16	push (4,4, dir 4)	(4,4) dir 3 (SE) → (5,5)	… · 5,3,2 · 4,4,4	6
17	Exit found	(5,5) dir 3 (SE) → (6,6) = X	stack + (5,5) + (6,6)	6

Table 8.2: Trace of `path`. The "Move" column gives the first open direction found from the current cell and the cell reached.

The first steps advance straight ahead. At `(1,1)` directions 1, 2, and 3 run into walls; direction 4 (south) opens onto `(2,1)`. In this move `(1,1)` is pushed with dir 5: if we return, the search will continue from the southwest. From `(2,1)` the search goes southeast, from there northeast, and from there north. The algorithm enters the corridor along the top edge of the maze through cells `(1,3)`, `(1,4)`, and `(1,5)`.

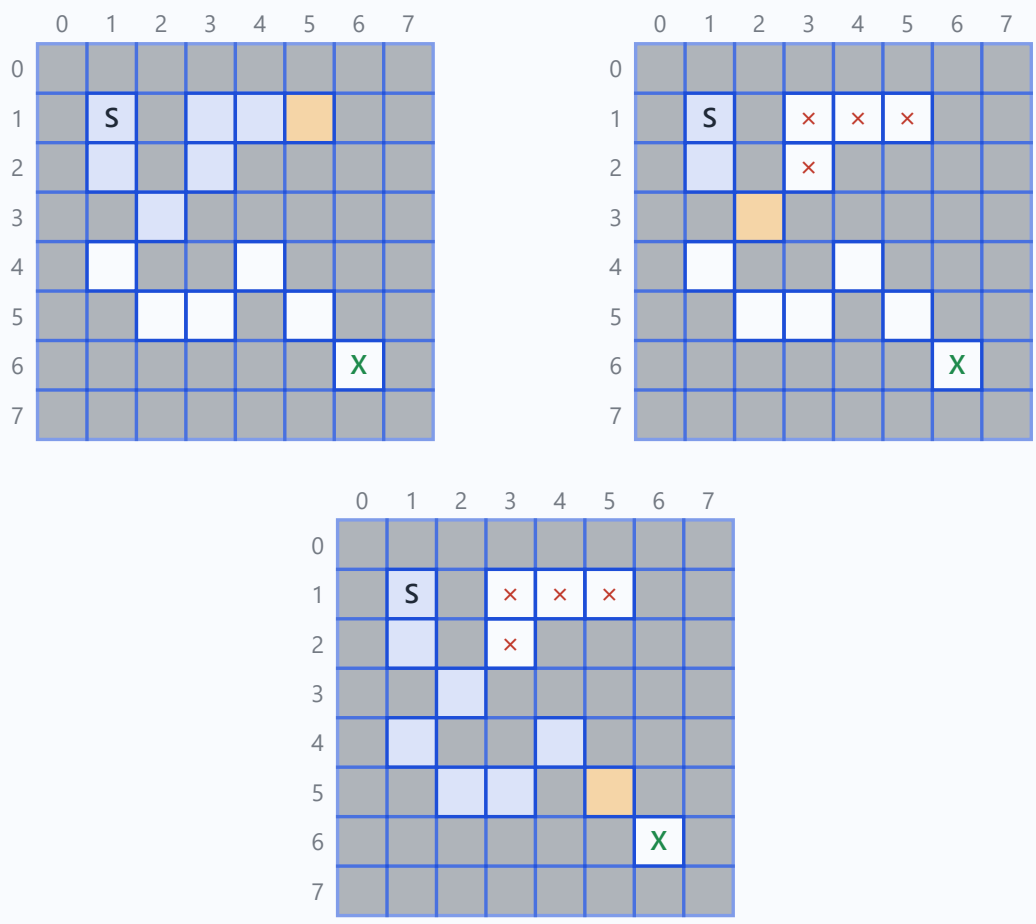


Figure 8.3: Three moments of the trace. On the left the algorithm has reached `(1,5)`; none of this cell's eight neighbors is open and unvisited. In the middle, after four steps back, the search has returned to `(3,2)` and continues from dir 2. On the right the exit has been found; the path consists of the positions on the stack, `(5,5)`, and `(6,6)`.

`(1,5)` is a dead end. Seven of its eight neighbors are walls and one (`(1,4)`) has been visited. The inner loop ends with `dir == 8` and the outer loop pops `(1,4, dir 3)` from the stack. This is one step back. At `(1,4)` the remaining directions from 3 on are tried: southeast `(2,5)` and south `(2,4)` are walls, southwest `(2,3)` and west `(1,3)` have been visited, and northwest is a wall. No direction opens up; the search goes back again. The same happens for `(1,3)` and `(2,3)`. After four steps back, the search arrives at `(3,2)` and continues from dir 2.

Here the value of the direction field shows. At `(3,2)` directions 0 and 1 have been tried before; direction 1 had in fact led to the dead-end corridor. The value 2 in the record prevents these two directions from being tried again. Directions 2, 3, and 4 are walls; direction 5 (southwest) opens onto `(4,1)`. From here the path runs toward the exit through a single corridor: `(4,1)` → `(5,2)` → `(5,3)` → `(4,4)` → `(5,5)`. At `(5,5)`, direction 3 (southeast) hits the exit cell and the search ends.

The path found is: `(1,1)`, `(2,1)`, `(3,2)`, `(4,1)`, `(5,2)`, `(5,3)`, `(4,4)`, `(5,5)`, `(6,6)`. That is 9 cells in total. The first seven are read from the stack (`top = 6`, that is, 7 records), the eighth is the current position, and the ninth is the exit. Over the whole run, 11 pushes and 5 pops were made. The stack reached at most 7 records. The four cells in the dead-end corridor (`(2,3)`, `(1,3)`, `(1,4)`, `(1,5)`) were visited but are not part of the path. Apart from the exit, all 12 open cells were visited.

8.2.6 The worst case

The maze program is a **brute-force** search. It uses no intelligence: it does not look at which direction the exit lies in, and it always tries the directions in the same order. The eight-entry look-up table has shortened the code and made the job easier, but the search itself is blind. How much work such an algorithm does depends largely on the input. In some mazes the exit is found in a few steps; in others nearly the whole maze is explored.

To state the worst-case cost of an algorithm, it is not enough to compute an upper bound; one must also show that an input that actually reaches that bound exists. This input is called the **worst-case input**. In the maze, what determines the worst-case input is the order in which directions are scanned. In every cell the algorithm looks north first, then northeast, and so on. If the first path tried in this order is very long, the algorithm follows it to the end.

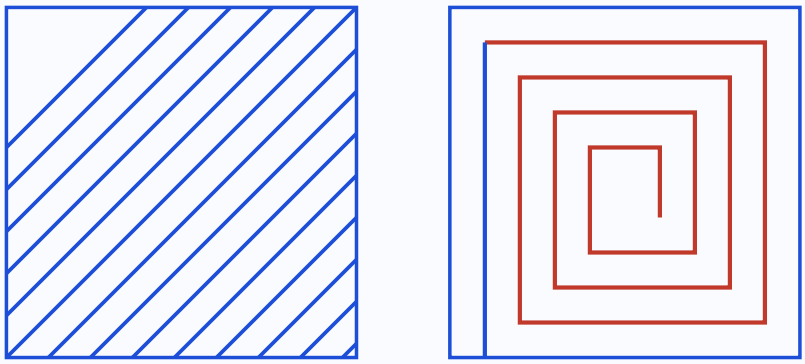


Figure 8.4: Two extreme inputs. In Input 1 the first path tried is long but correct. In Input 2 the first path tried (the red spiral) is long and wrong; the short path to the exit (blue) is tried only after the spiral has been explored in full.

Figure 8.4 compares the two inputs.

- **Input 1**: The first path tried is long but correct; it leads to the exit. The algorithm never goes back on this path. Only pushes are made, no pops. The amount of work is proportional to the length of the path.
- **Input 2**: The first path tried is long and wrong; it curls into the maze like a spiral and ends in a dead end. The algorithm follows this path to the end, and the stack fills with one record for every cell of the spiral. Once the dead end is reached, the stack is emptied one record at a time; at each step back the remaining directions of that cell are tried too. Only when the whole spiral is exhausted does the search return to the start and try the short, correct path. The solution is reached after everything except the correct path has been explored.

Input 2 is the worst-case input: it makes the algorithm do all the work it can possibly do. Every open cell is visited once, pushed onto the stack once, and popped once. This analysis also shows a principle. Before judging an algorithm "slow" or "fast", one must examine how it behaves on which input. It is not right to pass judgment on an algorithm that has not been analyzed.

8.2.7 Complexity and comparison with a queue

Let us find the cost of `path` by counting. Let the size of the maze be $m \times p$.

- **Time.** A cell is pushed onto the stack only when moving from it to an unvisited cell, and thanks to `mark` each cell is visited at most once. The total number of pushes is therefore at most the number of open cells. The number of pops cannot exceed it either. In every cell the inner loop tries at most eight directions; thanks to the direction field, directions tried earlier are not tried again on a return. The total number of direction trials for one cell is therefore at most 8, plus one round for each pop. The result: $O(8 \cdot m \cdot p) = O(m \cdot p)$. On the worst-case input this bound is reached, so the worst case is $\Theta(m \cdot p)$.

- **Memory.** The stack grows to at most the number of open cells: $O(m \cdot p)$. The `mark` matrix also holds $(m + 2)(p + 2)$ cells, which is $O(m \cdot p)$.

This search with a stack is an example of **depth-first search** (DFS). One goes in one direction and descends as deep as possible; one returns only at a dead end. This search has an important weakness: **the path found is not necessarily the shortest**. The algorithm returns the first path it finds. Because of the order in which directions are tried, that path may wind around a distant corner of the maze. In this maze the search ran into the dead end `(2,3)` → `(1,3)` → `(1,4)` → `(1,5)`, came back, and found a path of 9 cells.

If a **queue** is used instead of the stack, the character of the search changes completely. A queue works by the first-in-first-out (FIFO) rule. The start cell is put into the queue. At each step a cell is taken from the front of the queue and all of its unvisited open neighbors are added at the rear. The cells are thus expanded in order of their distance from the entrance: first those at distance 1, then those at distance 2, and so on. This method is called **breadth-first search** (BFS).

Definition 8.3

Depth-first search (DFS) is a search that goes to a neighbor, descends as deep as possible, and at a dead end returns to the latest fork; it is done with a stack or with recursion. **Breadth-first search** (BFS) is a search that visits nodes in order of their distance from the start; it is done with a queue. If all steps cost the same, the path by which BFS first reaches a cell is the **shortest path** to that cell.

Why BFS guarantees the shortest path can be explained briefly. The cells in the queue are always ordered by distance: first all those at distance d, then those at distance $d + 1$. When a cell is reached for the first time, a distance is recorded for it and the cell never enters the queue again. If a cell at distance d could be reached by a shorter path, the previous cell on that path would have left the queue earlier and the cell would have been reached before. The first arrival is therefore always by a shortest path.

In this maze BFS also finds the 9-cell path. That is, the two searches gave a path of the same length here; the BFS in the verification program computed it as 9 cells. The reason is that the maze is narrow: there is a single corridor to the exit. In general, the path found by DFS can be much longer. BFS has a price too: to be able to reconstruct the path, one must store from which cell each cell was reached. The time and memory cost of the two searches are of the same order: $O(m \cdot p)$. The complete code that finds the shortest path with BFS is given in the exercise solutions.

The person best placed to examine the code of these two algorithms is the one who wrote them. Someone else cannot analyze exactly what the code does without tracing it step by step. Code-trace tables are important for this reason: they give the author a chance to test the code and make its behavior visible to the reader.

8.3 Expression notations

One of the oldest and most common applications of the stack is the evaluation of arithmetic and logical expressions. When a compiler reads the line `x = a / b - c + d * e - a * c`, it has to decide in which order to compute this expression. This section defines the problem and introduces the three notations on which the solution rests.

8.3.1 The expression evaluation problem

Evaluation of expressions means computing the value of an expression while obeying operator precedence and associativity. An expression consists of two kinds of elements. **Operators** are symbols that specify the operation to perform, such as `+`, `-`, `*`, and `/`. **Operands** are the values the operation is applied to: variables such as `a` and `b`, or constants such as `4` and `2`. Parentheses are part of the expression as well and change the order of computation.

Let us see with an example why the problem is hard. Let `a = 4`, `b = 2`, `c = 2`, `d = 3`, `e = 3`. The expression `a / b - c + d * e - a * c` reads in mathematics as follows: first the divisions and multiplications, then the additions and subtractions from left to right. That is, `((4 / 2) - 2) + (3 * 3) - (4 * 2)`. This equals *2 − 2 + 9 − 8 = 1*. Someone who evaluates the same expression from left to right, ignoring precedence, would get *((((4/2) − 2) + 3) · 3 − 4) · 2 = 10*. The correct result is 1.

A human looks at the expression as a whole and picks out the multiplications first. A computer reads the expression from left to right, character by character. When it sees an operator it cannot apply it yet, because a higher-precedence operator may follow on its right. In `a + b * c`, computing `a + b` when the `+` is seen would be wrong; `b * c` is needed first. Parentheses complicate matters further: in the expression `a / (b - c) + d * (e - a) * c`, the subtractions inside the parentheses must be done first.

The solution compilers have used for years has two stages:

1. The expression is converted to **postfix** notation, taking operator precedence and parentheses into account. In this notation there are no parentheses and no precedence; the operators are lined up in the order in which they will be applied.
2. The postfix expression is evaluated from left to right in **a single pass**.

Both stages are done with a stack. This is not a new invention. Many people have solved this problem many times, and a well-known solution exists. On meeting a problem, one should first look for such known solutions and find the fastest way. Only if the problem is truly unsolved does discovery begin.

8.3.2 Precedence and associativity

Two rules determine which operator in an expression is applied first.

Definition 8.4

Operator precedence is the ranking that determines which of different operators is applied first: in C, `*` comes before `+`. **Associativity** determines whether operators of the same precedence are applied from left to right or from right to left: `a - b - c` means `(a - b) - c`, because `-` associates from left to right.

Table 8.3 summarizes the part of C's precedence table needed for these notes. The rows are given in order of decreasing precedence from top to bottom.

Precedence	Operators	Associativity
highest	`() [] -> .`	left to right
	unary `- ! ++ -- * &`	right to left
	`* / %`	left to right
	`+ -`	left to right
	`< <= > >=`	left to right
	`== !=`	left to right
	`&&`	left to right
lowest	`||`	left to right

Table 8.3: Operator precedence in C (summary). Upper rows are applied first.

When reading the table, precedence decides first; if precedence is equal, associativity decides. Parentheses override both: the part inside parentheses is always computed first. The unary operators in the second row (minus sign, logical not, increment and decrement, dereference and address-of) associate from right to left. For example, `*p++` means `*(p++)`. The table does not contain all of C's operators; shift, bitwise, conditional, and assignment operators are not included here.

Example 8.1

Let us parenthesize the expression `a / b - c + d * e - a * c` according to the rules. First the operators with the highest precedence: `/` and the two `*`. These become `(a / b)`, `(d * e)`, and `(a * c)`. Three additions/subtractions remain: `(a/b) - c + (d*e) - (a*c)`. All have the same precedence and associate from left to right. First the left `-` is applied, then `+`, and finally the right `-`. The result: `(((a / b) - c) + (d * e)) - (a * c)`.

The algorithms in these notes are limited to the five binary operators (`+ - * / %`) and parentheses. Unary minus, comparison, and logical operators do not change the method; only the precedence table grows. The table in the book by Horowitz, Sahni, and Anderson-Freed (Figure 3.12) contains all of these operators.

Common mistake

Forgetting associativity and evaluating operators of the same precedence from right to left. `8 - 4 - 2` from left to right is *(8 − 4) − 2 = 2*. Evaluated from the right, it gives *8 − (4 − 2) = 6*. For addition and multiplication this mistake does not change the result, because these operations are associative; for subtraction and division it does.

8.3.3 Infix, prefix, and postfix

An expression can be written in three forms, depending on where the operator stands relative to its operands. The notations are called *infix*, *prefix*, and *postfix*; these short names are used in the slide titles and function names (`postfix` , `prefix_eval`).

Definition 8.5

Infix notation: the operator is written between its two operands, `a + b` . To evaluate it, parentheses and precedence must be taken into account. **Prefix notation** (Polish notation): the operator is written before its operands, `+ a b` . **Postfix notation** (reverse Polish notation): the operator is written after its operands, `a b +` . In prefix and postfix notation no parentheses are needed; the expression is evaluated without looking at precedence.

The name "Polish notation" comes from the Polish logician Jan Łukasiewicz, who proposed prefix notation for logical formulas in the 1920s. Postfix notation is its reverse. Reverse Polish notation was once also used in some calculators: the user first entered two numbers and then pressed the operation key.

Table 8.4 gives the equivalents of four expressions in the three notations. Every row in the table was produced with the `postfix` and `prefix` functions of the verification program.

No	Prefix	Infix	Postfix
1	`*34`	`3*4`	`34*`
2	`+2*34`	`2+3*4`	`234*+`
3	`-*+abc/da`	`(a+b)*c-d/a`	`ab+c*da/-`
4	`**/a+-bcd-eac`	`(a/(b-c+d))*(e-a)*c`	`abc-d+/ea-*c*`

Table 8.4: The same expression in three notations.

The second row of the table shows how precedence disappears in postfix notation. In `2+3*4` the multiplication must be done first. Its postfix equivalent is `234*+` . Someone reading from the left sees `2` , `3` , and `4` first, then sees the operator `*` and applies it to the last two numbers, that is, to `3` and `4` . Then `+` adds `2` and `12` . There is no need to know the precedence: the order of the operators is the order of application itself.

In the third row the parentheses disappear. In `(a+b)*c` the addition must be done first. In the postfix equivalent, `+` is written before `*`: `ab+c*`. In postfix notation the position of the operators takes over the function of parentheses.

The fourth row is more complex. Because `b-c+d` associates from left to right, it is `(b-c)+d` and its postfix equivalent is `bc-d+`. The division puts `a` in front of it: `abc-d+/`. Then it is multiplied by `(e-a)` (`ea-*`) and finally multiplied by `c` (`c*`). Because the two multiplications associate from left to right, the left one is done first.

8.3.4 Converting by hand

There is a simple three-step way to convert an infix expression to postfix by hand. Table 8.5 applies these steps to the expression `a / b - c + d * e - a * c`.

Step	Expression
Start	`a / b - c + d * e - a * c`
1. Fully parenthesize	`((((a / b) - c) + (d * e)) - (a * c))`
2. Move the operator to its right parenthesis	`((((a b /) c -) (d e *) +) (a c *) -)`
3. Delete the parentheses	`a b / c - d e * + a c * -`

Table 8.5: Converting by hand: parenthesize, move, delete.

1. **Full parenthesization.** According to precedence and associativity, every operation is put into its own parentheses (Example 8.1). Every operator gets exactly one pair of parentheses.
2. **Moving.** Each operator is moved to the right closing parenthesis of its own pair. The pair `(a / b)` becomes `(a b /)`.
3. **Deleting.** The parentheses are deleted. The remaining sequence is the postfix expression: `a b / c - d e * + a c * -`.

For prefix notation, in the second step the operator is moved not to the right closing parenthesis but to the **left opening parenthesis** of its own pair. The prefix equivalent of the same expression is `- + - / a b c * d e * a c`.

An important observation lies in the result of the conversion: **the order of the operands does not change**. In the infix expression the operands are in the order `a b c d e a c`; in the postfix expression they are in the same order. Only the positions of the operators change. This observation is the basis of the stack algorithm. An operand can be written directly to the output the moment it arrives. Operators, on the other hand, wait on a stack until the right moment comes, that is, until the higher-precedence operations that follow them are finished.

Common mistake

Skipping the full parenthesization step when converting by hand and working only with the parentheses that appear in the expression. In `a / b - c` there are no parentheses at all, and so no closing parenthesis to move to. All operations must first be parenthesized as `((a / b) - c)`. Otherwise the result turns into a random arrangement.

8.3.5 Postfix to infix with a stack

The conversion in the opposite direction, that is, obtaining a fully parenthesized infix expression from a postfix expression, is a good first step for seeing how the stack works. The rule is short. The expression is read from left to right:

- If an operand arrives, it is pushed onto the stack.
- If an operator arrives, two items are popped from the stack. The first one popped is the right operand and the second is the left operand. The two are combined by putting the operator between them and enclosing the result in a pair of parentheses. The resulting string is pushed back onto the stack.
- When the expression ends, a single item remains on the stack: the fully parenthesized infix expression.

The example in the slides is the expression `42/2-33*+42*-`. This is the postfix expression obtained from `a / b - c + d * e - a * c` by substituting `a = 4`, `b = 2`, `c = 2`, `d = 3`, `e = 3`. Table 8.6 gives the stack after every step.

Token	Action	Stack
`4`	operand, onto the stack	4
`2`	operand, onto the stack	4 · 2
`/`	2 and 4 are popped, `(4/2)` is pushed	(4/2)
`2`	operand, onto the stack	(4/2) · 2
`-`	`((4/2)-2)` is pushed	((4/2)−2)
`3`, `3`	two operands onto the stack	((4/2)−2) · 3 · 3
`*`	`(3*3)` is pushed	((4/2)−2) · (3*3)
`+`	`(((4/2)-2)+(3*3))` is pushed	(((4/2)−2)+(3*3))
`4`, `2`	two operands onto the stack	(((4/2)−2)+(3*3)) · 4 · 2
`*`	`(4*2)` is pushed	(((4/2)−2)+(3*3)) · (4*2)
`-`	final combination	((((4/2)−2)+(3*3))−(4*2))

Table 8.6: Converting the expression `42/2-33*+42*-` to infix. The stack is written from left to right, bottom to top.

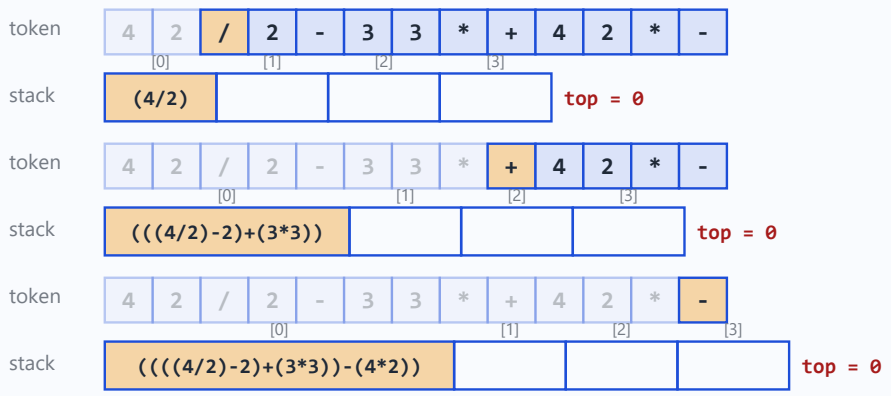


Figure 8.5: Three moments of the postfix-to-infix conversion. Top: the first `/` has been processed and `(4/2)` sits at the bottom of the stack. Middle: `+` has been processed; there is a single string on the stack. Bottom: the last `-` has been processed; the only item on the stack is the result.

The state of the stack at the two `*` steps is interesting. After the first `*` is processed, there are two items on the stack: `((4/2)-2)` and `(3*3)`. So cell `[1]` does not stay empty; `(3*3)` sits there and `top = 1`. After the second `*`, the same cell holds `(4*2)` this time. The stack never exceeds three items. In this expression the deepest moment is when the pair `3 3` or `4 2` is pushed onto the stack.

When this method is written in C, the stack holds strings rather than characters. Every combination requires allocating memory for a new string and freeing the two parts that are no longer needed. For this reason `malloc` and `free` must be used carefully. The complete code is given in the exercise solutions.

8.4 Evaluating a postfix expression

Evaluating a postfix expression is the second stage of the two-stage solution, but it is the easier one to understand. Let us therefore examine it first. The rule is almost the same as the rule for converting postfix to infix. The only difference is that the stack holds numbers instead of strings.

- If an operand arrives, its value is pushed onto the stack.
- If an operator arrives, two values are popped from the stack, the operation is performed, and the result is pushed onto the stack.
- When the expression ends, the single value left on the stack is the result.

8.4.1 The token: `get_token`

The evaluator reads the expression character by character. Each read yields a **token**. A token is the smallest meaningful piece of an expression: an operator, a parenthesis, an operand, or the end of the expression. In real compilers a token may consist of several characters; `123` or `total` is a single

token. To keep things simple, these notes assume that every operand is a single-digit number. Every character is then a token.

```
typedef enum { lparen, rparen, plus, minus,
    times, divide, mod, eos, operand } precedence;

precedence get_token(const char *expr, char *symbol, int *n) {
    *symbol = expr[(*n)++];
    switch (*symbol) {
        case '(':  return lparen;
        case ')':  return rparen;
        case '+':  return plus;
        case '-':  return minus;
        case '/':  return divide;
        case '*':  return times;
        case '%':  return mod;
        case '\0': return eos;
        default:   return operand;
    }
}
```

The first part of the code defines an **enumeration** type (`enum`). The declaration `typedef enum { … } precedence;` produces nine named integer constants: `lparen` is 0, `rparen` is 1, `plus` is 2, …, `operand` is 8. These constants label the kind of token. In the rest of the code, a readable name such as `plus` is used instead of the character `'+'` . Labeling with `typedef enum` makes the code easier to work with: readability improves, and the compiler can warn about labels forgotten in a `switch` .

The function `get_token` takes three parameters. `expr` is the expression being evaluated. `symbol` is the address where the character read is stored; the function thus both returns the kind of the token and passes the character itself to the caller. `n` is the address of the position currently being read in the expression. The line `*symbol = expr[(*n)++]` reads the character at the position `n` points to and advances the position by one. The next call reads the next character. Then the `switch` determines the kind of the character. Operator and parenthesis characters return their own labels, `'\0'` returns the end of the expression (`eos` , end of string), and everything else returns the label `operand` .

The verification program had the expression `"(7%3)"` read token by token and obtained the kinds `lparen` (0), `operand` (8), `mod` (6), `operand` (8), `rparen` (1), and `eos` (7) in turn.

An operand is a character; to compute with it, it must be converted to a number. The way to do this is the expression `symbol - '0'` . In C, a character is really a small integer that holds its code. In the ASCII table the digits are arranged consecutively: the code of `'0'` is 48, of `'1'` 49, of `'2'` 50. Hence `'2' - '0'` gives *50 − 48 = 2*. Subtracting `'0'` from a digit character gives its numeric value.

It was also possible to keep the expression in a global array named `expr`. This is how it is in the book by Horowitz and colleagues: the function `get_token` takes only two parameters and reads `expr` from the global variable. The version in the slides takes the expression as a parameter. This small difference is the first step of the "write it like a component" principle discussed in the next section.

8.4.2 `eval`: the global version

The first version of the routine that evaluates a postfix expression uses a global stack and a global expression, as in the book:

```
float eval(void) {
    char symbol;  precedence token;
    float op1, op2;
    int n = 0, top = -1;
    token = get_token(expr, &symbol, &n);
    while (token != eos) {
        if (token == operand)
            push(&top, symbol - '0');
        else {
            op2 = pop(&top);   /* right operand first */
            op1 = pop(&top);
            push(&top, apply(token, op1, op2));
        }
        token = get_token(expr, &symbol, &n);
    }
    return pop(&top);   /* the one left is the result */
}
```

The expression itself (`expr`) and the stack array (`stack`) are global; `top` is local and is passed to the stack functions by address. The flow of the routine is as follows:

- `n = 0` marks the start of the expression and `top = -1` the empty stack. The first token is read before the loop.
- The loop runs as long as the token is not `eos`. At the end of each round the next token is read. This is the pattern "read first, then process in the loop and read the next one".
- If the token is an operand, its numeric value, obtained with `symbol - '0'`, is pushed onto the stack.
- If the token is an operator, two values are popped from the stack. **The first one popped is `op2`, the right operand.** Then the left operand `op1` is popped. The function `apply` performs the operation and the result is pushed onto the stack.
- When the loop ends, the single value left on the stack is returned.

The function `apply` separates the operator kinds with a `switch`:

```
float apply(precedence token, float op1, float op2) {
    switch (token) {
        case plus:   return op1 + op2;
        case minus:  return op1 - op2;
        case times:  return op1 * op2;
        case divide: return op1 / op2;
        case mod:    return (int)op1 % (int)op2;
        default:     return 0;   /* not an operator: never reached */
    }
}
```

The order in which operands are popped is the most important detail of this routine. In a postfix expression the left operand is pushed onto the stack first and the right operand afterward. Because the stack returns the last item pushed first, the first value popped is the right operand. In the expression `6 2 -`, 6 is pushed first and then 2. When `-` arrives, the first value popped is 2 (`op2`) and the second is 6 (`op1`). The result is *op1 − op2 = 6 − 2 = 4*. Had the operands been popped in the opposite order, *2 − 6 = −4* would have been obtained. For addition and multiplication the order does not change the result, but for subtraction, division, and remainder it does.

The expression `(int)op1 % (int)op2` in the `mod` case comes from a rule of C. The stack holds `float` values, because the result of a division may be fractional. The `%` operator, however, cannot be used with `float`; it is defined only for integers. The two operands are therefore converted to `int` first. If the remainder of fractional values is needed, the function `fmod` of the standard library is used.

The `default` branch in `apply` is also necessary. If a `switch` over an enumeration type does not cover some of the labels, the compiler gives a `-Wswitch` warning. Here `lparen`, `rparen`, `eos`, and `operand` never reach this function; the `default` branch says so explicitly and silences the warning.

This routine was run in the verification program with the global `expr = "42/2-33*+42*-"` and returned the value 1. In the first version in the slides, `get_token` was called with two arguments, in the form `get_token(&symbol, &n)`. This call does not compile against the three-parameter definition of `get_token` ("too few arguments"). The correct call is `get_token(expr, &symbol, &n)`, which also passes the global `expr`.

8.4.3 Writing it like a component

The global version works, but it has a serious design problem. The routine depends on the outside world:

```
/* global version: external dependencies */
char expr[] = "42/2-33*+42*-";
float stack[MAX_SIZE];
float eval(void);
```

`eval` takes no parameters; it learns which expression to evaluate from the global `expr` and where to write from the global `stack`. This dependency has three consequences. First, two expressions cannot be evaluated at the same time: to evaluate the second one, the contents of `expr` must be changed. Second, the routine cannot be moved to another program; every program that wants to use it must define global variables of the same names. Third, the routine has side effects: while running, it changes an array that other parts of the program also use. A bug in one part of the program can show up in an entirely different place.

In the version written like a component, everything is a parameter or a local variable:

```
/* component: all parameters/locals */
float eval(const char *expr);
/* the stack is a local array in eval */
/* in main: */
float a = eval("23*54*+9-");
float b = eval("42/2-33*+42*-");
```

The expression comes in as a parameter; the stack is a local array inside `eval` and is set up anew at every call. The same routine works anywhere, with any expression, without side effects. The two calls in `main` do not affect each other at all. This version is used in the code traces.

This principle holds not only for this routine but for every function one writes. Functions should be written like a **component**: they should take everything they need as parameters and give their result as a return value. Uses such as global variables or `#define` constants read from inside the function create external dependencies. In a project worked on by a team, everyone writes independent pieces like this. The code of a programmer who says "put my variables above `main`, that is how I work" cannot be merged with other people's code. A personal style is an obstacle to joint work.

The complete code of the component version is:

```c
float eval(const char *expr) {
  float y[MAX_SIZE], op1, op2;  int top = -1;
  for (int n = 0; expr[n] != '\0'; n++) {
    char c = expr[n];
    if (isdigit(c)) y[++top] = c - '0';
    else {
      op2 = y[top--];  op1 = y[top--];
      switch (c) {
        case '+': y[++top] = op1 + op2; break;
        case '-': y[++top] = op1 - op2; break;
        case '*': y[++top] = op1 * op2; break;
        case '/': y[++top] = op1 / op2; break;
      }
    }
  }
  return y[top];
}
```

The code is shorter than the global version. Instead of `get_token`, the `for` loop walks directly over the characters of the expression; the loop stops at the character `'\0'`. `isdigit(c)` (`<ctype.h>`) tests whether the character is a digit. The stack operations are written as array expressions rather than function calls: `y[++top] = …` is a push and `y[top--]` is a pop. The line `op2 = y[top--]; op1 = y[top--];` again pops the right operand first. This time the `switch` is over the character itself and covers the four operators. `MAX_SIZE` is the size of the local stack; it must be larger than the number of operands in the expression.

8.4.4 Code trace: `eval("23*54*+9-")`

The first example is `23*54*+9-`. This is the postfix equivalent of the infix expression `2*3 + 5*4 - 9`. Its value should be *6 + 20 − 9 = 17*. Table 8.7 gives the contents of the stack after each character is processed.

`n`	`c`	**Action**	**Stack**	`top`
0	`'2'`	`'2' - '0'` = 2 onto the stack	2	0
1	`'3'`	3 onto the stack	2 · 3	1
2	`'*'`	`op2` = 3, `op1` = 2 → 2 * 3 = 6	6	0
3	`'5'`	5 onto the stack	6 · 5	1
4	`'4'`	4 onto the stack	6 · 5 · 4	2
5	`'*'`	`op2` = 4, `op1` = 5 → 5 * 4 = 20	6 · 20	1
6	`'+'`	`op2` = 20, `op1` = 6 → 6 + 20 = 26	26	0
7	`'9'`	9 onto the stack	26 · 9	1
8	`'-'`	`op2` = 9, `op1` = 26 → 26 − 9 = 17	17	0

n	c	Action	Stack	top
9	'\0'	the loop ends, y[0] is returned	17	0

Table 8.7: Trace of `eval("23*54*+9-")`. The stack is written from bottom to top.

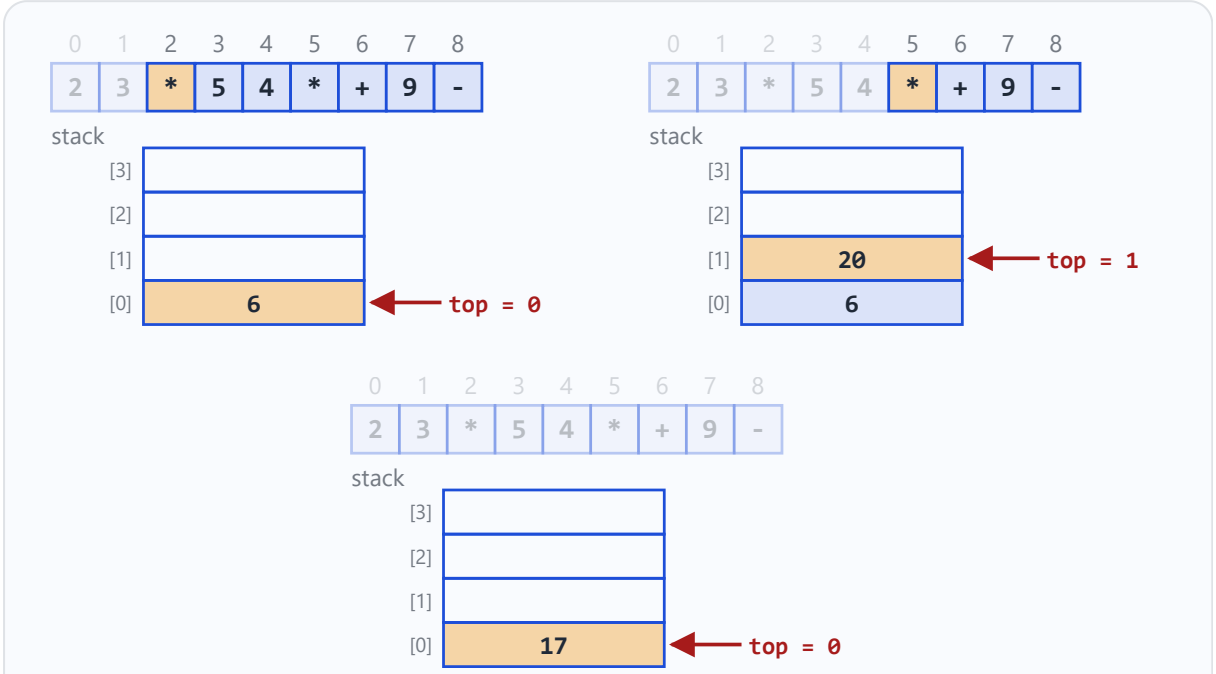


Figure 8.6: Three moments of the trace of `eval("23*54*+9-")`. In the first drawing the first `*` has been processed: 6 is on the stack. In the second the second `*` has been processed: 6 and 20 are on the stack, `top = 1`. In the third the expression has ended: the only value on the stack is 17.

The deepest moment of the stack in the trace is `n = 4`: three values (6, 5, 4) are on the stack at the same time. This is the moment when the product `5*4` is waiting to be added to `2*3`. In a postfix expression the stack stores the intermediate results that are waiting.

The cost is clear. Let the length of the expression be *n*. Each character is read once. For an operand one push is made; for an operator two pops, one operation, and one push are made. Each of these is *O(1)*. The total time is *Θ(n)*. In the worst case the stack holds all the operands at the same time (for example `1234+++`); the extra memory is *O(n)*.

8.4.5 Code trace: `eval("42/2-33*+42*-")`

The second example is the postfix equivalent of the expression that introduced the expression evaluation problem: `42/2-33*+42*-`. The expected value is 1. The infix equivalent of this expression was already found with the stack: `((((4/2)-2)+(3*3))-(4*2))`. Now the same stack works with numbers instead of strings.

c	Action	Stack
`'4'`, `'2'`	4 and 2 onto the stack	4 · 2
`'/'`	`op2` = 2, `op1` = 4 → 4 / 2 = 2	2
`'2'`	2 onto the stack	2 · 2
`'-'`	`op2` = 2, `op1` = 2 → 2 − 2 = 0	0
`'3'`, `'3'`	3 and 3 onto the stack	0 · 3 · 3
`'*'`	`op2` = 3, `op1` = 3 → 9	0 · 9
`'+'`	`op2` = 9, `op1` = 0 → 0 + 9 = 9	9
`'4'`, `'2'`	4 and 2 onto the stack	9 · 4 · 2
`'*'`	`op2` = 2, `op1` = 4 → 8	9 · 8
`'-'`	`op2` = 8, `op1` = 9 → 9 − 8 = 1	1

Table 8.8: Trace of `eval("42/2-33*+42*-")`.

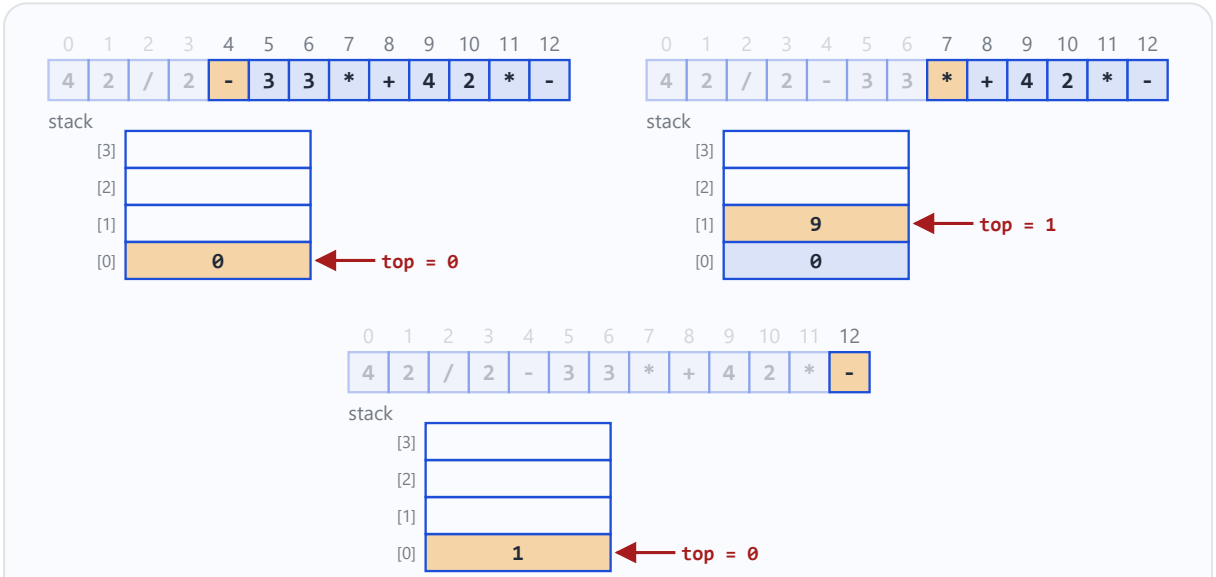


Figure 8.7: Three moments from the trace of `eval("42/2-33*+42*-")`. In the first drawing, after the first `-`, the stack holds 0. In the second, `3*3` has been computed and 9 sits on top of the 0. In the third, the last `-` has been processed and the result is 1.

This trace shows why the wrong "left-to-right" result of the first comparison does not appear. When the `+` operator arrives, the stack holds 0 and 9; that is, the product `3*3` has been computed before the addition. It is the conversion stage that knows the precedence. The evaluator merely applies the operators in the order in which they arrive.

The verification program evaluated these two expressions and the expression `623+-382/+*` from the exercise; the results are 17, 1, and 7. The program's dump of the intermediate stack is identical to every row in the tables.

8.4.6 Malformed expressions

In a well-formed postfix expression, exactly one value remains on the stack at the end of the evaluation. This rule can also be used as a validity test. Keep a counter through the expression: an operand increases it by one and a binary operator decreases it by one (it takes two values and puts back one). If the expression is valid, the counter is at least 2 whenever an operator arrives and is exactly 1 at the end.

- **If an operand is missing** (for example `2*`), there is a single value on the stack when the operator arrives. The short code in the slides executes `y[top--]` twice and, after `top` has dropped to −1, goes on to read `y[-1]`. This is a read outside the array and is undefined behavior.
- **If an operator is missing** (for example `234+`), two values remain on the stack when the expression ends. The code returns the top value and does not notice the error.

A robust routine should check `top >= 0` before every pop and at the end check that `top == 0`. The `pop` function in the slides takes over this check by printing an error message on an empty stack. Division by zero must also be handled separately: with `float` division the result is infinity or NaN (not a number), while for the `%` operation computing a remainder with a zero divisor is undefined behavior.

> **Common mistake**
>
> Popping `op1` and `op2` in the wrong order: `op1 = y[top--]; op2 = y[top--];`. With this code the expression `62-` gives −4 instead of 4, and `82/` gives 0.25 instead of 4. Test expressions that contain only addition and multiplication do not catch this bug; one must test with an expression that contains subtraction or division.

8.5 Converting infix to postfix

The first stage of the solution is to convert the infix expression to postfix notation. As observed in the hand conversion, the order of the operands does not change; only the positions of the operators change. The algorithm uses this observation directly: operands are written to the output the moment they arrive, while operators wait their turn on a stack.

8.5.1 The algorithm and `isp` / `icp`

Precedence determines when an operator is written to the output. In `a + b * c`, the `+` enters the stack and waits. When `*` arrives, `+` is still on the stack. Because `*` has higher precedence, it must be applied first, that is, it must be written to the output before `+`. So `*` goes on top of `+`. Conversely, in `a * b + c`, when `+` arrives, `*` is on the stack. `*` has higher precedence and can now be applied; before `+` enters the stack, `*` is moved to the output.

The rules of the algorithm are as follows. The expression is read from left to right.

1. An **operand** is written directly to the output.
2. When an **operator** arrives, as long as the precedence of the operator on top of the stack is greater than or equal to that of the incoming operator, the top operator is moved to the output. Then the incoming operator is pushed. Moving it to the output on equality as well ensures left-to-right associativity.
3. A **left parenthesis** `(` always enters the stack. While on the stack it behaves like the operator with the lowest precedence; every operator can go on top of it.
4. When a **right parenthesis** `)` arrives, the operators on the stack are moved to the output until a `(` is seen. Then the `(` is discarded from the stack. Parentheses are never written to the output.
5. **When the expression ends**, all the operators left on the stack are moved to the output.

The two different behaviors of the left parenthesis cannot be expressed with a single precedence value. On arrival it must pop no operator from the stack; this means the highest precedence. While on the stack, every operator must be able to go on top of it; this means the lowest precedence. The solution is to give every operator two precedences.

Definition 8.6

The **in-stack precedence** (`isp`) is the precedence an operator carries while waiting on the stack. The **incoming precedence** (`icp`) is its precedence at the moment it is read from the input. For an incoming operator `x`, the operator `t` on top of the stack is moved to the output as long as $isp(t) \geq icp(x)$.

Token	`isp` (in the stack)	`icp` (incoming)
`(`	0	20
`* / %`	13	13
`+ -`	12	12
`#` (bottom marker)	−1	—

Table 8.9: Precedence values. A larger number means higher precedence.

The numbers in the table themselves do not matter; only the order among them does. Horowitz and colleagues use values such as 12 and 13 to conform to C's full precedence table. The incoming precedence of `(` is 20 and no `isp` value reaches 20; therefore nothing comes off the stack when `(` arrives. Its in-stack precedence is 0 and the incoming precedence of every operator is greater than 0; therefore every operator goes on top of `(`. At the same time no operator can pop `(` from the stack, because $0 \geq 12$ is never true. Only `)` removes `(`.

The `#` at the bottom of the stack is a sentinel. Its in-stack precedence is −1; it is never greater than or equal to the incoming precedence of any operator. The loop "move the top operator to the output" therefore stops by itself when it reaches `#`. There is thus no need to check separately in the loop

condition whether the stack is empty. It is the same idea as the wall frame around the maze.

In the verification program the precedences were defined with the following two small functions:

```
int isp(char c) {           /* in-stack precedence */
  switch (c) {
    case '(': return 0;
    case '*': case '/': case '%': return 13;
    case '+': case '-': return 12;
    default: return -1;     /* '#': stack bottom */
  }
}
int icp(char c) {           /* incoming precedence */
  if (c == '(') return 20;
  return isp(c);
}
```

8.5.2 The `postfix` routine

```
void postfix(const char *e, char *out) {
  char y[MAX] = {'#'};  int top = 0, k = 0;
  for (; *e; e++) {
    if (isalnum(*e)) out[k++] = *e;
    else if (*e == ')') {
      while (y[top] != '(')
        out[k++] = y[top--];
      top--;                /* drop '(' */
    } else {
      while (isp(y[top]) >= icp(*e))
        out[k++] = y[top--];
      y[++top] = *e;
    }
  }
  while (top > 0) out[k++] = y[top--];
  out[k] = '\0';
}
```

- `y` is the operator stack. The declaration `char y[MAX] = {'#'}` writes `#` into the first cell (the other cells are zeroed). `top = 0` points to the bottom marker; that is, the stack is logically empty but there is a sentinel at the bottom. `k` is the position in the output string where the next character will be written.
- The loop `for (; *e; e++)` advances the pointer to the end of the expression, to `'\0'`. Because the parameter `e` is a copy, advancing it does not affect the caller's string.
- `isalnum(*e)` tests for a letter or digit. An operand is written directly to `out`.

- If `)` arrives, the inner loop moves operators to the output until the top of the stack is `(`. Then `top--` discards the `(`; it is not written to the output.
- For every other character (the four operators, `%`, and `(`), the loop "take while greater or equal" runs. Then the incoming character is pushed onto the stack. For `(` the loop never runs, because *icp('(') = 20*.
- After the loop, as long as `top > 0`, that is, until the `#` at the bottom is reached, the remaining operators are moved to the output. Finally the output string is closed with `'\0'`.

The function writes its result into the `out` array supplied by the caller. The caller must allocate this array large enough. Because a postfix expression contains no parentheses, it is never longer than the infix expression; space for the length of the expression plus `'\0'` is enough.

8.5.3 Code trace: `a*(b+c)-d`

Let us run the routine on `a*(b+c)-d`. The `#` is always at the bottom of the stack; it is not shown in the table or the figures.

Token	Decision	Stack	`top`	Output
`a`	operand → output	(empty)	0	a
`*`	*isp(#) = −1 < icp(*) = 13* → push	*	1	a
`(`	*isp(*) = 13 < icp(() = 20* → push	* (	2	a
`b`	operand → output	* (	2	ab
`+`	*isp(() = 0 < icp(+) = 12* → push	* (+	3	ab
`c`	operand → output	* (+	3	abc
`)`	pop up to `(` : `+` to the output, `(` discarded	*	1	abc+
`-`	*isp(*) = 13 ≥ icp(−) = 12* → `*` to the output; *isp(#) < 12* → push `-`	−	1	abc+*
`d`	operand → output	−	1	abc+*d
end	remaining `-` to the output	(empty)	0	abc+*d−

Table 8.10: Trace of `postfix("a*(b+c)-d")`. The `#` at the bottom is not shown in the stack column; `top` is the variable of the code.

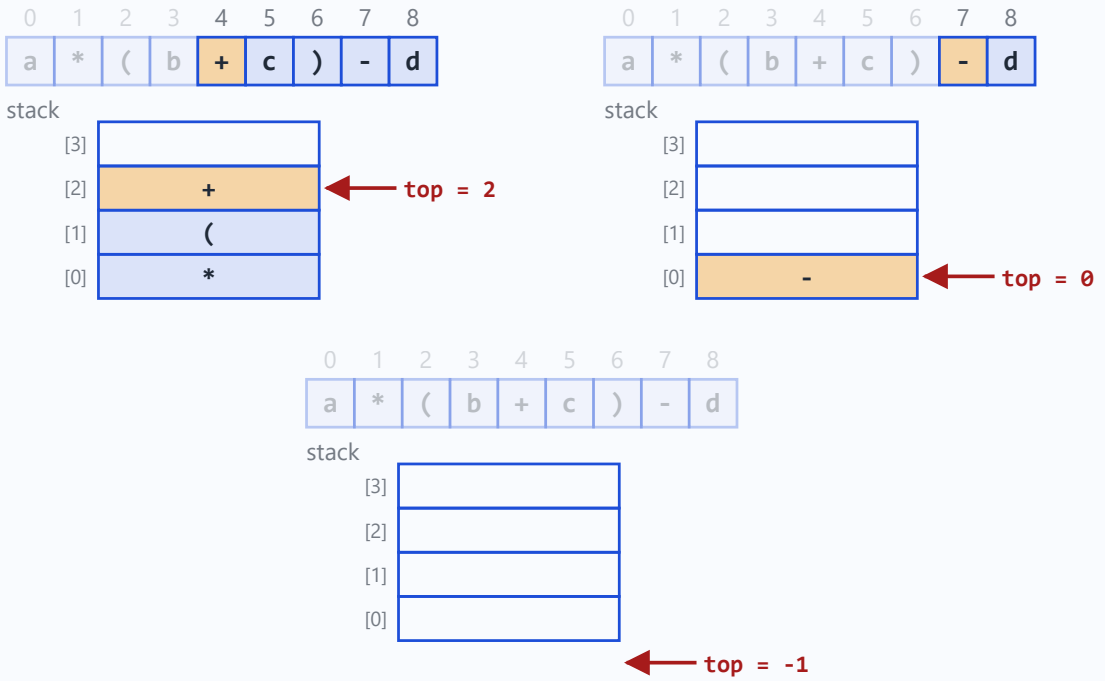


Figure 8.8: Three moments of the conversion of `a*(b+c)-d`. In the first drawing `+` has gone on top of the `(` on the stack. In the second, `-` has arrived, moved `*`, whose precedence is equal or higher, to the output, and entered the stack itself. In the third, the input has ended and the result is `abc+*d-`.

The key moment of the trace is `)`. The operation `b+c` inside the parentheses cannot be completed until the closing parenthesis arrives; `+` therefore waits on the stack. When `)` arrives, `+` is moved to the output. The contents of the parentheses thus pass to the output so as to be applied before the outer `*`. The result is `abc+*d-`. This expression reads "first `b+c`, then multiply by `a`, then subtract `d`".

8.5.4 Equal precedence: `a*(b+c)*d`

The second example differs by a single character: `*` instead of `-`. The trace is the same up to `)`. The difference shows when the second `*` arrives. The first `*` is on top of the stack and *isp(*) = 13 ≥ icp(*) = 13*. Equality is also enough to move it to the output. The first `*` is moved to the output and the second `*` enters the stack. The result is `abc+*d*`.

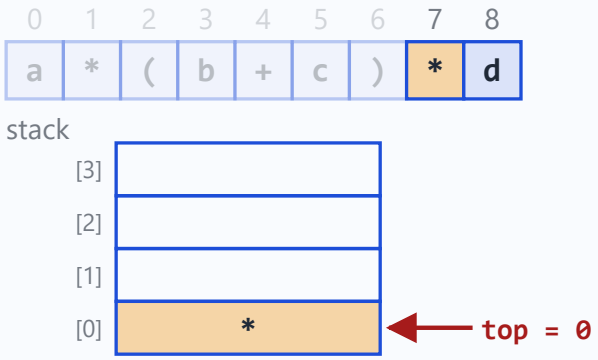


Figure 8.9: The moment the second `*` arrives in the conversion of `a*(b+c)*d` : the first `*` , of equal precedence, has been moved to the output and the second `*` has entered the stack.

Moving to the output on equality means **left-to-right associativity**. The expression `a*(b+c)*d` means `(a*(b+c))*d` : the left multiplication is done first. If the left multiplication is written to the output first, it is applied first in the postfix expression too. Because multiplication is associative, the result does not change here, but for subtraction it does. With the correct rule, `a-b-c` is converted to `ab-c-` , that is, `(a-b)-c` . Code that does not move on equality (one that uses `>`) produces `abc--` . This means `a-(b-c)` . For *a = 9*, *b = 5*, *c = 2*, the correct result is 2 and the result of the wrong code is 6. The verification program ran both versions and found these two values.

Common mistake

Writing the comparison as `isp(y[top]) > icp(*e)` . The code gives the right result on most test expressions; it goes wrong only on consecutive operators of equal precedence. The bug can therefore go unnoticed for a long time. Expressions such as `a-b-c` and `a/b/c` must always be tested.

8.5.5 Trace table: `a+b*c`

The last example in the slides is `a+b*c` . Table 8.11 follows the format in the book by Horowitz and colleagues: the stack cells `[0]` and `[1]` are separate columns and the bottom marker is not counted. The `top` values in the table are therefore one less than the variable in the code: in the code `#` sits in cell `y[0]` and `+` is written to `y[1]` .

Token	[0]	[1]	top	Output
a			−1	a
+	+		0	a
b	+		0	ab
*	+	*	1	ab
c	+	*	1	abc

Token	[0]	[1]	`top`	Output
end			−1	abc*+

Table 8.11: Trace table of `a+b*c` → `abc*+` (without counting the bottom marker).

When `*` arrives, `+` is on top of the stack. Because *isp(+) = 12 < icp(*) = 13*, `+` is not moved to the output; `*` goes on top of it. When the input ends, the stack is emptied from the top: first `*` comes out, then `+`. The operator with higher precedence is applied first.

A frequently asked question about this table is: "Isn't the last operand `c` printed to the output?" It is. The `postfix` routine writes operands to the output the moment they arrive; `c` has also gone to the output before the last row. At the end only the operators waiting on the stack are emptied.

8.5.6 Cost

Let the length of the expression be n. The outer loop processes each character once. At first glance the inner loops seem to make the cost quadratic: a single `)` or a single operator can move many items from the stack to the output. But each operator enters the stack at most once and leaves it at most once. The total number of rounds of all the inner loops therefore cannot exceed the number of operators that entered the stack, that is, it is less than n. This kind of counting is called **aggregate analysis**: the total cost of all steps is computed, not that of a single step. The total time is $\Theta(n)$. The stack and the output array use $O(n)$ extra memory.

Conversion and evaluation together are also linear: first the postfix expression is produced in $\Theta(n)$ and then evaluated in $\Theta(n)$. When parsing an expression, compilers usually merge these two steps or build an expression tree instead of a postfix expression, but the basic idea is the same.

8.6 Prefix notation

Prefix notation is the mirror image of postfix notation. The operator comes before its operands. The methods of conversion and evaluation are also similar to those for postfix; the difference lies in the direction from which the expression is read.

8.6.1 Infix to prefix

An easy way to convert an infix expression to prefix is to use the `postfix` algorithm we already have. The method consists of four steps. Table 8.12 applies the steps to the expression `(a+b)*c-d/a`.

Step	Expression
Start	`(a+b)*c-d/a`
1. Reverse	`a/d-c*)b+a(`

Step	Expression
2. Swap `(` and `)`	`a/d-c*(b+a)`
3. Convert to postfix (do not pop on equality)	`ad/cba+*-`
4. Reverse again	`-*+abc/da`

Table 8.12: Infix to prefix by reversal.

The logic of the method is this. When the expression is reversed, the operand on the left of each operator moves to the right and the one on the right moves to the left. When the postfix of the reversed expression is found, the operator is written after its operands. When the result is reversed again, the operator goes in front of its operands and the operands return to their original order. Reversal also flips the parentheses: `(a+b)` becomes `)b+a(`. This is why the parentheses are swapped in the second step.

There is an important change in the third step: on equal precedence nothing is **popped** from the stack; the comparison is made with `>`. In the reversed expression the associativity is reversed as well. `a-b-c`, that is, `(a–b)–c`, becomes `c-b-a` when reversed. For this expression to convert to the correct prefix, the left `-` must stay on the stack and the right `-` must go on top of it. With this rule the verification program found `--abc` for `a-b-c`, that is, `–(–(a, b), c)`; this is the correct prefix equivalent.

The C code of the method can be written as follows (it uses the same `isp` / `icp` functions):

```
/* infix -> prefix: reverse, swap parentheses,
   convert to postfix without popping on equality (>), reverse again */
void prefix(const char *e, char *out) {
  char t[MAX], y[MAX] = {'#'};
  int n = strlen(e), top = 0, k = 0;
  for (int i = 0; i < n; i++) {        /* reverse */
    char c = e[n - 1 - i];
    t[i] = c == '(' ? ')' : c == ')' ? '(' : c;
  }
  t[n] = '\0';
  for (char *p = t; *p; p++) {
    if (isalnum(*p)) out[k++] = *p;
    else if (*p == ')') {
      while (y[top] != '(') out[k++] = y[top--];
      top--;
    } else {
      while (isp(y[top]) > icp(*p))    /* > : no pop on equal */
        out[k++] = y[top--];
      y[++top] = *p;
    }
  }
  while (top > 0) out[k++] = y[top--];
  out[k] = '\0';
  for (int i = 0, j = k - 1; i < j; i++, j--) {   /* reverse again */
    char tmp = out[i]; out[i] = out[j]; out[j] = tmp;
  }
}
```

The result is the same as the third row of Table 8.4. The function produced the prefix equivalent in the table for all four expressions. The reversals, the stack operations, and the conversion apply a constant number of operations to each character of the expression; the total time is $\Theta(n)$.

There is also a direct way. The expression is scanned from right to left. Operands are added to the front of the output; operators are handled with the same stack rule, again without popping on equality. The role of the parentheses is also swapped: when reading from right to left, `)` is seen first and enters the stack, and when `(` is seen, the stack is emptied. Both ways are $O(n)$.

8.6.2 Evaluating a prefix expression

A prefix expression is evaluated by reading it from right to left, with the same stack technique as postfix evaluation:

```
float prefix_eval(const char *e) {
  float y[MAX], op1, op2;  int top = -1;
  for (int i = strlen(e) - 1; i >= 0; i--) {
    if (isdigit(e[i])) y[++top] = e[i] - '0';
    else {
      op1 = y[top--];  op2 = y[top--];
      y[++top] = apply(e[i], op1, op2);
    }
  }
  return y[top];
}
```

`i` starts at the last character and decreases to 0. A digit is pushed onto the stack. When an operator arrives, two values are popped and the function `apply` performs the operation. This function separates the four operators with a `switch`:

```
float apply(char op, float op1, float op2) {
  switch (op) {
    case '+': return op1 + op2;
    case '-': return op1 - op2;
    case '*': return op1 * op2;
    case '/': return op1 / op2;
    default:  return 0;
  }
}
```

The crucial difference in the code is the order of popping. In a prefix expression, the left operand is written first and the right operand after it, immediately to the right of an operator. Reading from right to left, the right operand is seen first and pushed first. The left operand is pushed afterward and stays on top. Therefore **the first value popped is the left operand**: `op1 = y[top--]; op2 = y[top--];`. In postfix evaluation, the first value popped was the right operand.

The expression `strlen(e) - 1` contains a detail about types. `strlen` returns the unsigned `size_t`. On an empty string, `0 - 1` turns into a very large number in unsigned arithmetic. Because the code assigns this result to `int i`, in practice −1 is obtained and the loop never runs. Had the loop variable been made `size_t`, the condition `i >= 0` would always be true and the loop would run forever.

The example in the slides is `-+*23*549`. This is the prefix equivalent of the expression `2*3+5*4-9`; its value should be 17, as in the postfix example.

`i`	`e[i]`	**Action**	**Stack**
8	`9`	onto the stack	9
7	`4`	onto the stack	9 · 4
6	`5`	onto the stack	9 · 4 · 5

i	e[i]	Action	Stack
5	*	op1 = 5 (left), op2 = 4 → 20	9 · 20
4	3	onto the stack	9 · 20 · 3
3	2	onto the stack	9 · 20 · 3 · 2
2	*	op1 = 2, op2 = 3 → 6	9 · 20 · 6
1	+	op1 = 6, op2 = 20 → 26	9 · 26
0	-	op1 = 26, op2 = 9 → 26 − 9 = 17	17

Table 8.13: Trace of `prefix_eval("-+*23*549")`. The expression is read from right to left.

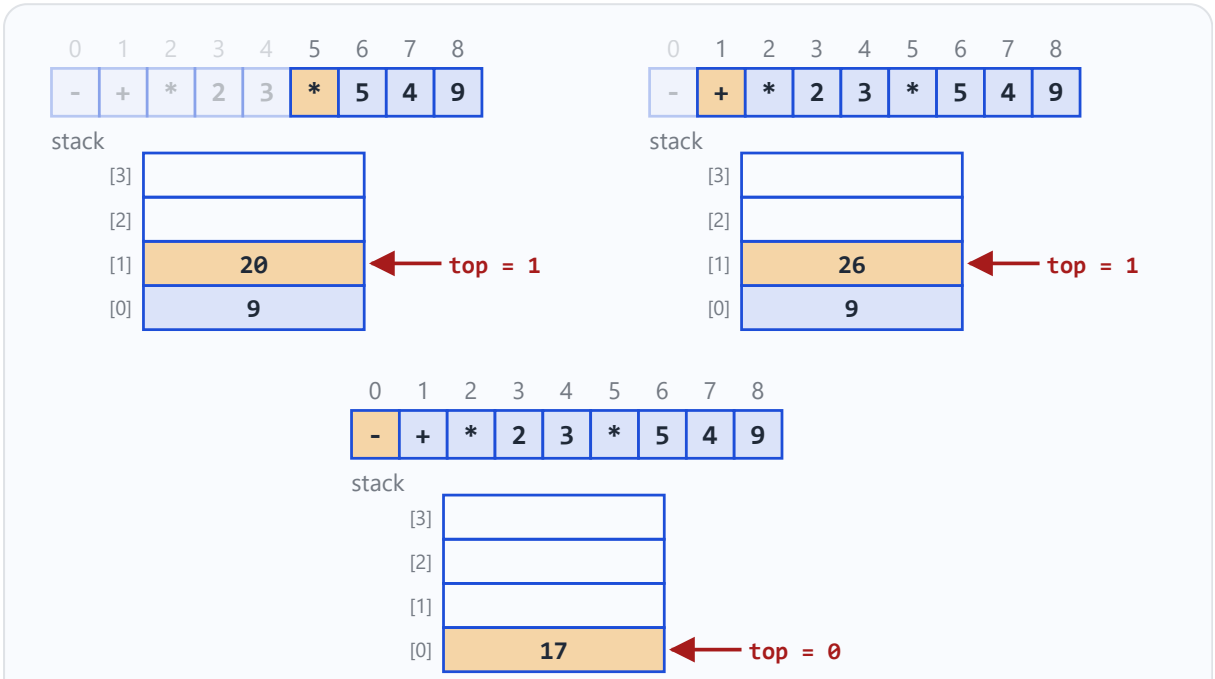


Figure 8.10: Three moments of prefix evaluation. In the first drawing, the first `*`: the first value popped, 5, is the left operand, and the result is 20. In the second, `+`: 6 + 20 = 26. In the third, `-`: 26 − 9 = 17.

In the last step the popping order decides the result. On the stack 9 is at the bottom and 26 on top. The first value popped, 26, is the left operand and the result is *26 − 9 = 17*. Had the postfix rule been applied here, *9 − 26 = −17* would have been obtained. The verification program found 4 for `-62` and *(6 − 2) · 3 = 12* for `*-623`.

8.6.3 The other conversions

The three notations and evaluation form a four-cornered diagram. Figure 8.11 gives this diagram. Green arrows show the routes whose code is given in these notes, and red question marks show the conversions that remain to be designed.

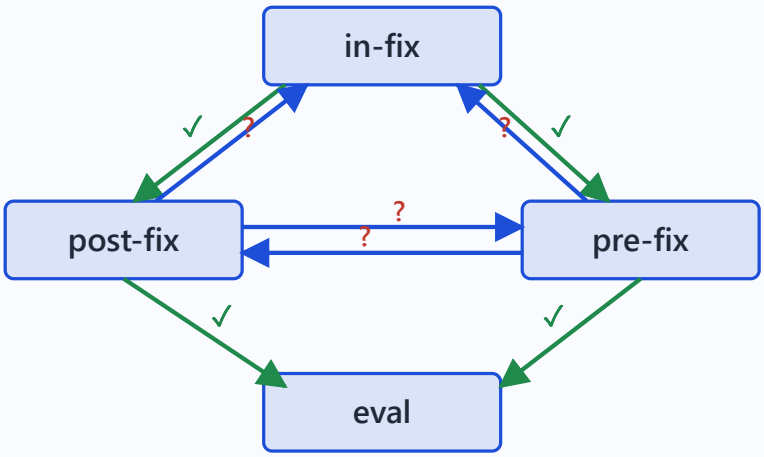


Figure 8.11: Conversions between notations. ✓: done in these notes (infix to postfix, infix to prefix by reversal, postfix and prefix evaluation). ?: conversions to be designed.

All the remaining conversions can be designed with the same stack idea:

- **Postfix to infix**: writing the hand conversion in C. The stack holds strings; the solution is given in the exercises.
- **Postfix to prefix**: the same as postfix to infix, only the combination is made in the form `op a b`. **Prefix to postfix**: the expression is read from right to left and the combination has the form `a b op`.
- **Infix to prefix directly**: the expression is scanned from right to left (described in the previous section).
- **Prefix to infix**: read from right to left; the first value popped is the left operand, and the combination has the form `(a op b)`.
- **Evaluating an infix expression directly**: two stacks are used, one for operands and one for operators. At the moment the conversion algorithm would write an operator to the output, that operator instead pops two values from the operand stack and pushes the result back. The value is thus computed directly without ever producing a postfix expression.

8.7 Parenthesis checking

The conversion from infix to postfix assumes that the parentheses in the expression are correctly formed. This assumption can be checked with a separate algorithm. **Parenthesis checking** (balanced parentheses) tests that every opening parenthesis is closed with the right type and in the right order. The problem is one of the cleanest applications of the stack.

Definition 8.7

The parentheses in a string are **balanced** if the following three conditions hold: every closing parenthesis has a partner that was opened before it and has not yet been closed; this partner is **the most recently opened** of the unclosed openers and is of the same type as the closer (`()` , `[]` , `{}`); and when the string ends, no unclosed opener remains.

The phrase "most recently opened" in the middle of the definition points to the stack. In the string `{a*(b+c)-[d]}` , the `(` was opened after the `{` and must be closed before it. If the openers are put on a stack, when a closer arrives its partner is always on top of the stack.

8.7.1 The `balanced` routine

```
bool balanced(const char *e) {
  char y[MAX];  int top = -1;
  for (; *e; e++) {
    if (*e=='(' || *e=='[' || *e=='{')
      y[++top] = *e;
    else if (*e==')' || *e==']' || *e=='}') {
      if (top < 0 || !matches(y[top], *e))
        return false;  /* extra/wrong */
      top--;
    }
  }
  return top == -1;       /* none left open? */
}
```

- If an opening parenthesis (`(` , `[` , `{`) arrives, it is pushed onto the stack.
- If a closing parenthesis arrives, two cases are checked. If the stack is empty (`top < 0`), this closer has no partner: an extra closer. If the stack is not empty, the code checks whether the opener on top matches the closer. If it does not, the types are mixed up: a wrong match. In both cases the function returns `false` at once. If they match, the opener is removed from the stack (`top--`).
- Characters that are not parentheses are skipped.
- If the stack is empty when the string ends (`top == -1`), every opener has been closed and the result is `true` . If it is not empty, there are unclosed openers.

The expression `top < 0 || !matches(y[top], *e)` in the condition relies on short-circuit evaluation. The `||` operator evaluates its left side first; if the left side is true, it does not look at the right side at all. When the stack is empty, `y[top]` , that is, `y[-1]` , is not read. Had the conditions been written in the opposite order, an empty stack would cause a read outside the array. The function `matches` is not given in the slides; in the verification program it was written as follows:

```
bool matches(char opener, char closer) {
  return (opener == '(' && closer == ')') ||
         (opener == '[' && closer == ']') ||
         (opener == '{' && closer == '}');
}
```

8.7.2 Code trace: `{a*(b+c)-[d]}`

Character	Action	Stack	top
`{`	opener → push	{	0
`a`, `*`	skipped	{	0
`(`	opener → push	{ (	1
`b`, `+`, `c`	skipped	{ (	1
`)`	matches the `(` on top → pop	{	0
`-`	skipped	{	0
`[`	opener → push	{ [	1
`d`	skipped	{ [	1
`]`	matches the `[` on top → pop	{	0
`}`	matches the `{` on top → pop	(empty)	−1
end	`top == -1` → `true`	(empty)	−1

Table 8.14: Trace of `balanced("{a*(b+c)-[d]}")`. Characters that are not parentheses are collected into a single row.

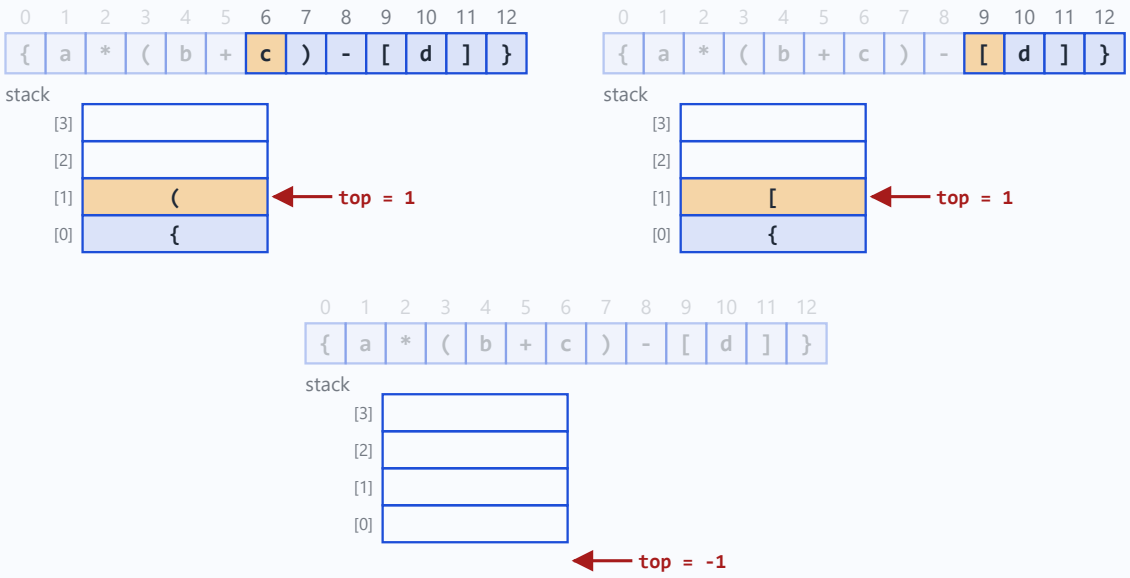


Figure 8.12: Three moments of parenthesis checking. In the first drawing `{` and `(` are on the stack. In the second, the `(` has been closed and `[` has been opened. In the third, the input has ended and the stack is empty: the string is balanced.

The stack held at most two openers. This is the depth of nesting: `(` and `[` are each inside `{` but independent of each other.

8.7.3 Three kinds of errors

An unbalanced string shows up in three different ways. Table 8.15 gives an example of each kind; all three examples gave `false` in the verification program.

Input	What happens?	Error
`[(a+b]*c)`	When `]` arrives, `(` is on top	Wrong match
`(a+b))`	When the second `)` arrives, the stack is empty	Extra closer
`((a+b)`	When the input ends, `(` is left on the stack	Unclosed opener

Table 8.15: Three kinds of errors in parenthesis checking.

For a single kind of parenthesis no stack is needed; a counter is enough. At an opener the counter increases by one and at a closer it decreases by one. If the counter never goes below zero and is zero at the end, the string is balanced. With several kinds of parentheses a counter is not enough. In the string `[(a+b]*c)` the numbers of openers and closers are equal, but the types are crossed. To catch this, one must answer the question "which was the most recently opened?" The stack holds this information.

This algorithm runs inside a great deal of software. Compilers check the parentheses and braces in source code this way. HTML and XML validators match opening and closing tags by the same method; there, tag names rather than parentheses are pushed onto the stack. A text editor highlighting the partner of a parenthesis when the cursor is on it is also an application of this algorithm.

Cost: if the length of the string is n, each character is examined once and each operation is $O(1)$; the time is $O(n)$. In the worst case all the characters are openers and all enter the stack; the extra memory is $O(n)$. If an error is found early, the function returns without reaching the end of the string.

8.8 Other applications of stacks and queues

The maze, expression evaluation, and parenthesis checking share a common idea: "the last one opened is the first one closed". Wherever this idea is seen, a stack can be used. Wherever "in order", that is, "first come, first processed", is seen, a queue is used. This section briefly introduces a few other applications of this principle.

8.8.1 Undo and redo

The **undo** and **redo** commands in text editors work with two stacks. When the user performs an operation, the operation is pushed onto the "undo" stack and the "redo" stack is emptied. The undo command takes the operation on top of the "undo" stack, applies its inverse, and puts it onto the

"redo" stack. The redo command takes the operation on top of the "redo" stack, applies it again, and puts it onto the "undo" stack.

For example, let the user perform the operations A, B, and C in turn. The "undo" stack is A · B · C. If undo is performed twice, first C and then B are undone. A remains on the "undo" stack and C · B on the "redo" stack, with B on top. If redo is performed once, B is applied again. If at this moment the user performs a new operation D, the C on the "redo" stack is deleted. A new branch has started and the old future cannot be returned to. The back and forward buttons of web browsers also work with the same two stacks.

8.8.2 Non-recursive tree traversal

Visiting the nodes of a binary tree in some order is called **traversal**. Traversal is naturally written recursively: in inorder traversal, the left subtree is traversed first, then the root is visited, then the right subtree is traversed. Recursion stores the pending work on the call stack. The same job can also be done with an explicit stack; this removes the overhead of recursive calls and puts the size of the stack under the programmer's control.

The following node type is used for the code: each node carries a character and two subtree pointers.

```
typedef struct node *tree_pointer;
typedef struct node {
  char data;
  tree_pointer left, right;
} node;

void inorder(tree_pointer p) {          /* non-recursive */
  tree_pointer y[MAX];  int top = -1;
  for (;;) {
    for (; p; p = p->left) y[++top] = p;  /* push while going left */
    if (top < 0) break;                  /* stack empty: done */
    p = y[top--];
    printf("%c", p->data);
    p = p->right;                        /* move to right subtree */
  }
}

void preorder(tree_pointer p) {           /* non-recursive */
  tree_pointer y[MAX];  int top = -1;
  if (p == NULL) return;
  y[++top] = p;
  while (top >= 0) {
    p = y[top--];
    printf("%c", p->data);
    if (p->right) y[++top] = p->right;     /* push right first, */
    if (p->left) y[++top] = p->left;       /* left stays on top */
  }
}

void postorder(tree_pointer p) {          /* with two stacks */
  tree_pointer y1[MAX], y2[MAX];  int t1 = -1, t2 = -1;
  if (p == NULL) return;
  y1[++t1] = p;
  while (t1 >= 0) {
    p = y1[t1--];
    y2[++t2] = p;                        /* order: root, right, left */
    if (p->left) y1[++t1] = p->left;
    if (p->right) y1[++t1] = p->right;
  }
  while (t2 >= 0) printf("%c", y2[t2--]->data);   /* reversed: left, right, root */
}
```

The function `inorder` performs inorder traversal. The inner `for` loop starts at `p`, descends by following left links, and pushes every node it passes onto the stack. When the left end is reached, the node on top of the stack is popped, printed, and the traversal moves to its right subtree. The traversal ends when the stack is empty and `p` is `NULL`. At every moment the stack holds the nodes on the path from the root to the current node whose right side has not yet been visited. The size of the stack therefore does not exceed the height of the tree.

`preorder` is simpler. The root is pushed onto the stack. In each round the node on top is popped and printed; then its right child and after it its left child are pushed. Because the left child stays on top, it is processed first. `postorder` is written with two stacks. The first stack transfers the nodes to the second in the order "root, right, left". When the second stack is emptied in reverse, the order "left, right, root" is obtained, that is, postorder traversal.

The verification program built the tree of the expression `(a+b)*c-d/a`: root `-`, left subtree `(a+b)*c`, right subtree `d/a`. The three traversals gave the outputs `a+b*c-d/a` (infix without parentheses), `-*+abc/da` (prefix), and `ab+c*da/-` (postfix) in turn. The three traversals of an expression tree are the three notations of the expression. Every one of the 9 `malloc` calls made to build and release the tree was matched by a `free`.

Cost: in a tree of n nodes, each node enters the stack once and leaves once; the time is $O(n)$. In inorder and preorder traversal the stack holds at most as many nodes as the height of the tree, h (the exact bound for preorder is of the order of $h + 1$); the extra memory is $O(h)$. In a balanced tree $h = O(\log n)$, and in a tree degenerated into a single branch $h = n - 1$. In postorder traversal with two stacks, the extra memory is $O(n)$ because the second stack holds all the nodes. **Level-order traversal**, which processes the tree level by level, is done with a queue: the root is put into the queue, in each step a node is taken from the front, and its children are added at the rear.

8.8.3 Next greater element

Suppose that for each element of an array we want to find the first greater element to its right. Two nested loops do this in $O(n^2)$. A stack does the same job in one pass:

```
/* first greater element to the right of each item; -1 if none */
void next_greater(const int a[], int n, int result[]) {
  int y[MAX], top = -1;                 /* indices on the stack */
  for (int i = 0; i < n; i++) {
    while (top >= 0 && a[y[top]] < a[i])
      result[y[top--]] = a[i];          /* a[i] answers the waiting item */
    y[++top] = i;
  }
  while (top >= 0) result[y[top--]] = -1; /* nothing greater to the right */
}
```

The stack holds the indices of the items that have not yet found their answer. The values of these items are in decreasing order from bottom to top; this is why the structure is called a **monotonic stack**. When a new item `a[i]` arrives, as long as the waiting item on top is smaller than it, `a[i]` is the answer for that item and the item is popped. Then `i` is pushed. When the array ends, the items remaining on the stack have no greater item to their right; their answers are −1.

For the array `{4, 5, 2, 25, 7, 3, 8}` the verification program gave the result `5 25 25 -1 8 8 -1`. For example, when 25 arrives, the indices of 5 and 2 are on the stack; both are smaller than 25, and the answer for both is 25. Because every index enters the stack once and leaves once, the total

time is *O(n)*; the extra memory is *O(n)* in the worst case (a decreasing array).

8.8.4 Sliding window maximum

Let a window of width *k* be slid from left to right over an array of *n* items, and let the largest item in the window be wanted at every position. Scanning each window separately takes *O(n · k)*. A **double-ended queue** (deque), that is, a queue in which items can be inserted and removed at both ends, brings this down to *O(n)*:

```
/* maximum of each window of width k; d: deque holding indices */
void sliding_max(const int a[], int n, int k, int result[]) {
  int d[MAX], front = 0, rear = -1;         /* d[front..rear] */
  for (int i = 0; i < n; i++) {
    if (front <= rear && d[front] <= i - k) front++;           /* left the window */
    while (front <= rear && a[d[rear]] <= a[i]) rear--;  /* drop smaller ones */
    d[++rear] = i;
    if (i >= k - 1) result[i - k + 1] = a[d[front]];      /* maximum at front */
  }
}
```

The deque holds indices, and their values are in decreasing order from front to rear. When a new item arrives, first the index that has fallen out of the window is dropped from the front. Then every index whose value is not greater than the new item is dropped from the rear: those items can never be the maximum as long as the new item stays in the window. The new index is added at the rear. The maximum of the window is always at the front. For the array `{1, 3, -1, -3, 5, 3, 6, 7}` and *k = 3*, the result is `3 3 5 5 6 7`. Each index is added once and dropped at most once; the time is *O(n)* and the extra memory is *O(k)*.

8.8.5 Shortest paths with a queue and other backtracking problems

If a queue is used instead of the stack in the maze, the first path found is the shortest. This is the basic property of breadth-first search, and beyond the maze it holds in every network in which the steps cost the same. A game character reaching its goal in the fewest moves, or the shortest chain of acquaintances between two people in a social network, is found by the same method.

The other classic examples of backtracking are also solved with a stack, or with recursion. The **Towers of Hanoi** problem keeps the pending subproblems on a stack while moving disks from one peg to another. The **n-queens** problem places *n* queens on a chessboard so that none threatens another. One queen is placed in each row; if a dead end is reached, the last queen placed is removed and the next column is tried. A **Sudoku** solver tries digits in the empty cells in turn and undoes the last attempt when a contradiction appears. All three carry the skeleton of the maze routine: a choice is pushed onto the stack and popped at a dead end. These problems are usually written recursively, but they can also be written with an explicit stack as in the maze.

8.9 Complexity summary

Table 8.16 gives the costs of the algorithms in these notes together. The justification of each row was explained in the relevant section.

Algorithm	Structure	Time	Extra memory
Maze (`path`)	stack	$O(m \cdot p)$	$O(m \cdot p)$
Maze shortest path	queue	$O(m \cdot p)$	$O(m \cdot p)$
Postfix / prefix evaluation	stack	$O(n)$	$O(n)$
Infix to postfix / prefix	stack	$O(n)$	$O(n)$
Parenthesis checking	stack	$O(n)$	$O(n)$
Non-recursive inorder	stack	$O(n)$	$O(h)$
Level-order traversal	queue	$O(n)$	$O(n)$
Next greater element	stack	$O(n)$	$O(n)$
Sliding window maximum	deque	$O(n)$	$O(k)$

Table 8.16: Complexity summary. n: expression length or number of nodes; $m \times p$: maze size; h: tree height.

All the linear times in the table rest on the same justification: each item enters the structure at most once and leaves at most once, and every entry and exit is $O(1)$. Nested loops do not break this bound, because the total number of rounds of the inner loops is limited by the number of items removed. This counting method is the tool used most often in analyzing stack and queue algorithms.

8.10 Common mistakes

1. **Popping `op1` and `op2` in the wrong order in postfix evaluation.** The first one popped is the right operand. If they are popped the other way round, `6 2 -` gives −4 instead of 4.
2. **Using the postfix order in prefix evaluation.** A prefix expression is read from right to left and the first one popped is the left operand.
3. **Not popping from the stack on equal precedence in the infix-to-postfix conversion.** Code that compares with `>` converts `a-b-c` to `abc--`, that is, to `a-(b-c)`.
4. **Leaving `(` on the stack or writing it to the output when `)` arrives.** If `(` is not discarded, it stays on the stack and blocks the operators that follow; when the stack is emptied at the end, it also slips into the output.
5. **Forgetting the `mark` in the maze.** The algorithm goes back and forth between two cells, falls into an infinite loop, or overflows the stack.
6. **Not surrounding the maze with 1s.** The neighbors of cells on the edge fall outside the array; C does not check this and undefined behavior results.

7. **Reading the top item in parenthesis checking without checking whether the stack is empty.** In `(a+b))`, the second `)` reads `y[-1]` on an empty stack.
8. **Not checking for stack overflow.** In an array-based stack of fixed size, if `MAX` is exceeded, items are written outside the array. Either the size must be chosen safely for the input or a check must be made at every push.

8.11 Exercises and solutions

The solutions to the six exercises in the slides are given in detail below. All the code was compiled and run in the verification programs.

8.11.1 Converting an expression to postfix and prefix

The expression is `a/(b-c)+d*(e-a)*c`. Let us first convert by hand. The fully parenthesized form is `((a/(b-c)) + ((d*(e-a))*c))`. Division and the two multiplications take precedence over addition; the two multiplications associate from left to right.

- **Postfix**: every operator is moved to its right closing parenthesis. `(b-c)` → `bc-`, `a/…` → `abc-/`, `(e-a)` → `ea-`, `d*…` → `dea-*`, `…*c` → `dea-*c*`. Total: `abc-/dea-*c*+`.
- **Prefix**: every operator is moved to its left opening parenthesis. The result is `+/a-bc**d-eac`.

The same result is also found with the `postfix` routine. In brief: `a` to the output; `/` onto the stack; `(` onto the stack; `b`, `c` to the output; `-` on top of `(`; when `)` arrives, `-` to the output. When `+` arrives, because *isp(/) = 13 ≥ 12*, `/` goes to the output and `+` onto the stack. `d` to the output; `*` goes on top of `+`. `(e-a)` produces `ea-` in the same way. When the second `*` arrives, the first `*`, of equal precedence, is moved to the output. After `c` the input ends; the `*` and `+` on the stack pass to the output in turn. The `postfix` and `prefix` functions of the verification program also gave the results `abc-/dea-*c*+` and `+/a-bc**d-eac`.

8.11.2 Evaluating a postfix expression

The expression is `6 2 3 + - 3 8 2 / + *`. The stack changes from left to right as follows: 6; 6 · 2; 6 · 2 · 3; `+` → *2 + 3 = 5*, stack 6 · 5; `-` → *6 − 5 = 1*, stack 1; 1 · 3; 1 · 3 · 8; 1 · 3 · 8 · 2; `/` → *8 / 2 = 4*, stack 1 · 3 · 4; `+` → *3 + 4 = 7*, stack 1 · 7; `*` → *1 · 7 = 7*. The result is 7. The stack held at most four values. `eval("623+-382/+*")` also returned 7.

8.11.3 A C routine that converts postfix to infix

The stack now holds dynamically allocated strings instead of characters. The method is the same as the hand conversion from postfix to infix.

```
/* postfix -> fully parenthesized infix; the stack holds dynamic strings */
char *postfix_infix(const char *e) {
  char *y[MAX];  int top = -1;
  for (; *e; e++) {
    if (isalnum((unsigned char)*e)) {
      char *s = malloc(2);
      if (s == NULL) return NULL;
      s[0] = *e;  s[1] = '\0';
      y[++top] = s;
    } else {
      char *b = y[top--], *a = y[top--];       /* right operand first */
      char *s = malloc(strlen(a) + strlen(b) + 4);
      if (s == NULL) return NULL;
      sprintf(s, "(%s%c%s)", a, *e, b);
      free(a);  free(b);                       /* parts no longer needed */
      y[++top] = s;
    }
  }
  return y[top];             /* the caller must free it */
}
```

- When an operand arrives, a two-byte string is allocated: the character itself and `'\0'`. The address of the string is pushed onto the stack.
- When an operator arrives, two strings are popped; the first one popped, `b`, is the right operand. The size of the new string is the length of the two parts plus four bytes: two parentheses, the operator, and `'\0'`. `sprintf` joins the parts in the form `"(a op b)"`.
- After the parts have been copied into the new string they are no longer needed and are released with `free`. If this line is forgotten, two blocks leak at every operator.
- The return value is the last string on the stack. Releasing this string is the caller's job.

The verification program converted four expressions. For `42/2-33*+42*-` it produced `((((4/2)-2)+(3*3))-(4*2))`, and for `ab+c*da/-` it produced `(((a+b)*c)-(d/a))`. With wrapper functions that count `malloc` and `free` calls, 44 allocations and 44 releases were counted; there is no leak. The code also has a gap: if `malloc` fails, the function returns `NULL` but does not release the strings left on the stack. A robust version must empty the stack before returning. In terms of cost, each combination copies two parts; as the strings grow longer, the copying grows longer. In an expression of length n, the total copying is $O(n^2)$ in the worst case (for example, in a left-leaning chain such as `ab+c+d+…`, every step copies the whole earlier string).

8.11.4 Marking the path found in the maze

When the exit is found, the path consists of the positions on the stack, the last position, and the exit. Writing `2` into these cells makes the path visible in the maze:

```
/* called after path finds the exit: marks the path in the maze with 2 */
void mark_path(int top, short int row, short int col) {
  for (int i = 0; i <= top; i++)
    maze[stack[i].row][stack[i].col] = 2;
  maze[row][col] = 2;                          /* last position */
  maze[EXIT_ROW][EXIT_COL] = 2;                /* exit */
}
```

It is called at the end of `path` with the line `if (found) mark_path(top, row, col);`. The function does not change the stack; it only reads it. The rows of the maze then look as follows (1 wall, 0 unvisited path, 2 path found):

```
11111111
12100011
12101111
11211111
12112111
11221211
11111121
11111111
```

The cells `(1,3)`, `(1,4)`, `(1,5)`, and `(2,3)` in the dead-end corridor stay 0: they were visited but did not enter the path. Marking takes *O(top)* time, that is, time proportional to the length of the path.

8.11.5 Finding the shortest path in the maze with a queue

The `(row, col)` pairs of the cells are put into the queue. For every cell, the cell from which it was reached is stored in the `prev` matrix. When the exit is reached, this information is followed backward to build the path.

```
/* shortest path in the maze: queue + the cell each cell was reached from */
typedef struct { short int row, col; } cell;
int shortest_path(void) {
  cell q[64], prev[8][8];                /* 6x6 interior: at most 36 cells */
  int head = 0, tail = 0;
  short int seen[8][8] = {{0}};
  q[tail++] = (cell){1, 1};  seen[1][1] = 1;
  while (head < tail) {
    cell h = q[head++];                  /* take from the front */
    if (h.row == EXIT_ROW && h.col == EXIT_COL) {
      int len = 1;                       /* trace back from the exit */
      while (h.row != 1 || h.col != 1) {
        maze[h.row][h.col] = 2;          /* mark the path */
        h = prev[h.row][h.col];
        len++;
      }
      maze[1][1] = 2;
      return len;
    }
    for (int dir = 0; dir < 8; dir++) {
      int s = h.row + move[dir].vert;
      int t = h.col + move[dir].horiz;
      if (!maze[s][t] && !seen[s][t]) {
        seen[s][t] = 1;
        prev[s][t] = h;
        q[tail++] = (cell){s, t};        /* add at the rear */
      }
    }
  }
  return 0;                              /* no path */
}
```

- The queue is a simple array: `head` is the position to take from at the front and `tail` the position to write to at the rear. Because each cell enters the queue at most once, a circular structure is not needed; an array with as many places as the interior has cells is enough.
- A cell is marked (`seen`) when it enters the queue, not when it leaves. In this way the same cell cannot enter the queue twice from two different neighbors.
- If the cell taken from the front is the exit, the `prev` chain is followed back to the entrance. At every step the cell is marked with `2` and the path length grows by one.

In the verification program the function found the 9-cell path and marked the maze in the same way as the search with a stack; in this maze the shortest path is unique. At most 13 cells enter the queue. The time is $O(m \cdot p)$ and the extra memory (`prev` , `seen` , the queue) is $O(m \cdot p)$.

8.11.6 Non-recursive preorder and postorder traversal

The solution consists of the functions `preorder` and `postorder` in the section on non-recursive tree traversal. In preorder traversal the root is pushed onto the stack; in each round the node on top is popped and printed, and its right and then left child are pushed onto the stack. Postorder traversal is done with two stacks: the first stack produces the order "root, right, left" and the second stack reverses this order. It can also be written with a single stack; in that case, for each node a visit flag that says whether its right subtree has been traversed, or the address of the last node printed, is kept.

8.12 Summary

- The **stack** works by the last-in-first-out rule. In the array implementation, push is `y[++top] = x` and pop is `y[top--]`; both are $O(1)$.
- **Backtracking**: in the maze, every fork is pushed onto the stack; at a dead end an item is popped from the stack and the search returns to the last fork. The direction field in the record makes sure that directions already tried are skipped on a return. The wall frame around the maze removes the boundary check, and the `mark` matrix prevents infinite loops.
- The **worst case** is a long and wrong first path: the entire region except the correct path is explored. `path` uses $O(m \cdot p)$ time and memory; the path it finds is not necessarily the shortest. Breadth-first search with a queue guarantees the shortest path.
- **Notations**: infix notation needs precedence and parentheses; prefix and postfix expressions are evaluated without parentheses and without precedence. The order of the operands is the same in all three notations.
- **Postfix evaluation**: an operand is pushed onto the stack; at an operator, first `op2` (right) and then `op1` (left) are popped and the result is pushed onto the stack. $\Theta(n)$. In prefix evaluation the expression is read from right to left and the first one popped is the left operand.
- **Infix to postfix**: operands are written to the output; the operator on the stack is moved to the output as long as $isp \geq icp$. `(` has the highest precedence on arrival and the lowest on the stack; when `)` arrives, the stack is emptied down to `(`. Popping on equality gives left associativity. $\Theta(n)$.
- **Parenthesis checking**: an opener is pushed onto the stack, and a closer must match the opener on top; at the end the stack must be empty. Three kinds of errors: wrong match, extra closer, unclosed opener.
- **General principle**: a stack wherever "the last opened is the first closed" is seen, a queue wherever "in order" is seen. Undo/redo, non-recursive traversal, next greater element, and the sliding window are other applications of this principle.
- Functions should be written **like a component**: inputs come in as parameters, the result goes out as the return value, and no dependency on global variables is created.

8.13 Key terms

stack LIFO backtracking maze look-up table brute force worst case depth-first search (DFS) breadth-first search (BFS) shortest path expression evaluation operator precedence associativity infix prefix (Polish notation) postfix (reverse Polish notation) token enumeration (enum) in-stack precedence (isp) incoming precedence (icp) global variable balanced parentheses short-circuit evaluation undo deque (double-ended queue) expression tree call stack

8.14 Test Yourself

1. What are the postfix and prefix equivalents of the expression `a*b-(c+d)/e`?

   Answer

   The fully parenthesized form is `((a*b) - ((c+d)/e))`. Postfix: `ab*cd+e/-`. Prefix: `-*ab/+cde`. In the `postfix` routine, when `-` arrives, the `*` on the stack is moved to the output; `(` goes on top of `-`; when `)` arrives, `+` passes to the output; `/` goes on top of `-`. At the end `/` and `-` come out.

2. What is the value of the postfix expression `82/3-42*+`? How many values does the stack hold at most?

   Answer

   8 / 2 = 4, *4 − 3 = 1*, *4 · 2 = 8*, *1 + 8 = 9*. The result is 9. The stack holds at most three values: at the moment `4` and `2` are pushed, the stack holds 1 · 4 · 2.

3. What value does `prefix_eval("+*23-84")` return? What are `op1` and `op2` while `-` is processed?

   Answer

   From right to left: 4 and 8 are pushed. `-`: the first one popped, `op1` = 8, `op2` = 4, the result is 4. Then 3 and 2 are pushed; `*`: `op1` = 2, `op2` = 3, the result is 6. `+`: *6 + 4 = 10*. The result is 10.

4. What does `balanced("{[}]")` return? At which character, and why?

> **Answer**
>
> `false`. `{` and `[` are pushed onto the stack. When the third character `}` arrives, `[` is on top of the stack; `matches('[', '}')` is false. This is the wrong-match error. Since the numbers of openers and closers are equal, a single counter could not catch this error.

5. If the comparison in the `postfix` routine is written as `>` instead of `>=`, how is `a-b-c` converted? What are the values of the two results for `a = 9`, `b = 5`, `c = 2`?

> **Answer**
>
> When the second `-` arrives, the first `-` is not popped from the stack; the second `-` goes on top of it. The output is `abc--`, that is, `a-(b-c)`. Its value is *9 − 3 = 6*. The correct conversion, `ab-c-`, has the value *(9 − 5) − 2 = 2*.

6. Would the path found change if the direction field of the initial record in `path` were 0 instead of 1?

> **Answer**
>
> It would not. Direction 0 looks north from `(1,1)`, that is, at the wall `[0][1]` on the frame. This direction is tried, skipped with `++dir` because it is a wall, and the search continues from direction 1 in the same way. Only one unnecessary trial is added.

7. How would `a*(b+c)` be converted if `isp('(')` were 20 instead of 0?

> **Answer**
>
> When `+` arrives, *isp(() = 20 ≥ icp(+) = 12*, and `(` is moved to the output; then `*` is moved to the output as well (*13 ≥ 12*). A parenthesis gets mixed into the output and the order breaks. When `)` arrives, there is no `(` on the stack; the inner loop runs past the bottom marker and reads outside the array. This is why `(` must have the lowest precedence on the stack.

8. What is the postfix of the expression `(a+b)*(c-d)/e`? What is its value for `a = 5`, `b = 3`, `c = 4`, `d = 2`, `e = 1`?

> **Answer**
>
> Postfix: `ab+cd-*e/`. `*` and `/` have equal precedence and associate from the left; when `/` arrives, `*` is moved to the output. Value: *5 + 3 = 8*, *4 − 2 = 2*, *8 · 2 = 16*, *16 / 1 = 16*.

9. If a queue is used instead of a stack in the maze, why is the first path found the shortest? Why is a

cell that enters the queue marked as it enters?

Answer

The queue yields cells in order of their distance from the entrance: all cells at distance *d* are processed before those at distance *d + 1*. The path that first reaches a cell is therefore the shortest. If a cell were not marked on entering, another neighbor could add it to the queue a second time before it left the queue; the queue would grow needlessly and the `prev` information could be overwritten by a wrong path.

8.15 Further reading

- E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2nd ed., Sections 3.5 and 3.6. The maze routine, the `isp` / `icp` table, and expression evaluation are there in the form followed in these notes.
- R. Sedgewick, *Algorithms in C*, Parts 1–4, Section 4.3. Postfix evaluation and infix-to-postfix conversion are treated as the first applications of the stack abstract data type; the relationship between recursion and the stack is treated in Chapter 5.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 3rd ed., MIT Press, 2009, Chapters 10 and 22. The array implementation of the stack and queue and breadth-first and depth-first search are presented together with their correctness proofs.

8.16 References

1. E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2nd ed., Silicon Press, 2008. Sections 3.5 (maze) and 3.6 (expression evaluation).
2. R. Sedgewick, *Algorithms in C*, Parts 1–4, 3rd ed., Addison-Wesley, 1998. Sections 4.3 (postfix) and 5 (recursion and the stack).
3. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 3rd ed., MIT Press, 2009.

PART II · CHAPTER 9

The List ADT and Generalized Lists

What can be done with a list and how a list is stored in memory are two separate questions. These notes first answer the first question: they write the operations of the list abstract data type as a contract, with preconditions and postconditions. They then implement the same contract with an array and with a linked list and compare the two implementations by cost. In the second half the notion of a list is generalized. Generalized lists are introduced, whose elements may be not only single values but other lists as well. On this structure the routines for copying, equality, traversal, depth, and inversion are written recursively and traced step by step. The last sections cover how a tree can be stored as a single line of text, the problems caused by shared sublists, reference counting, and garbage collection.

LEARNING OUTCOMES

Contract
Define the list abstract data type by its operations with preconditions and postconditions; separate the interface from the implementation with an opaque type.

Comparison
Compare the array and linked implementations in terms of access, insertion, deletion, and memory; explain the amortized cost of growing a dynamic array.

Generalized list
Define lists that contain atoms and sublists, represent them in C with a tagged `union`, and draw their memory layout.

Recursive routines
Write copying, equality, traversal, depth, and inversion recursively, trace them, and justify their costs.

Tree representation
Convert a tree to a one-line linear form and explain the limit of this method for binary trees.

Sharing
Explain the pitfalls of shared and recursive lists, reference counting, and the idea of garbage collection.

9.1 The list abstract data type

When a program uses a list, it wants to know two things: which operations can be applied to the list, and what state the list will be in at the end of each operation. Whether the list lives in memory as an array or as nodes linked to one another is not the business of the code that uses it. This separation is one of the basic ideas of software engineering and is the subject of this section.

9.1.1 The linear list and the contract

> **Definition 9.1**
>
> A **linear list** is an ordered sequence of $n \geq 0$ elements: $L = (a_0, a_1, \ldots, a_{n-1})$. n is the **length** of the list; if $n = 0$, the list is empty. For every $i < n - 1$, a_i comes before a_{i+1}. The place of an element in the list, that is, i, is its **position**.

The definition speaks only of order. Whether the elements sit side by side in memory, or how one passes from one element to the next, is not in the definition. The days of the week, the student numbers of a class, or the lines of a text fit this definition. The jobs to be done with such lists are also clear: reading the i-th element, inserting an element at a position, deleting the element at a position, searching for a value, and learning the length.

A type that defines operations and their behavior while hiding the implementation is called an **abstract data type** (ADT). The **list ADT** is an abstract data type that offers insertion, deletion, and access by position. In C, the outward face of an abstract data type is usually declarations written in a **header file**. The slides give the interface of the list ADT as follows:

```c
typedef struct list List;          // opaque type

List  *list_create(void);
void   list_destroy(List *L);
int    list_length(const List *L);
bool   list_get(const List *L, int i, int *x);
bool   list_insert(List *L, int i, int x);  // 0<=i<=n
bool   list_delete(List *L, int i, int *x);  // 0<=i<n
int    list_find(const List *L, int x);    // -1 if absent
```

Each function's name says what it does. `list_create` builds an empty list and returns its address; `list_destroy` returns all the memory belonging to the list. `list_get(L, i, &x)` writes the i-th element into `x`, `list_insert(L, i, x)` inserts `x` at position i, and `list_delete(L, i, &x)` removes the i-th element and writes it into `x`. `list_find` returns the position of the sought value, or −1 if it is not found. Operations that return a value deliver it through a pointer parameter; this leaves the return value free to report whether the operation succeeded.

The name and parameters of a function are not enough by themselves. What should the call `list_insert(L, 7, x)` do on a list of length 3? The answers to such questions are the **precondition** and the **postcondition** written for every operation. The precondition is the state that must be true before the operation is called. The postcondition is the state that is guaranteed to be true at the end of the operation if the precondition was met. Together they form the **contract** of the operation. The contract for `list_insert` is this:

- **Precondition:** $0 \le i \le n$. $i = n$ means inserting at the end; the upper bound is therefore n, not $n - 1$.
- **Postcondition:** `x` is at position i, the length has become $n + 1$, and the old elements $a_i, \ldots, a_{n-1}$ have each moved one position to the right. That is, the old a_k is now at position $k + 1$. The elements before i stay in place.

For `list_delete` the precondition is $0 \le i < n$; deleting from an empty list is therefore invalid for every i. In the postcondition the length decreases by one and the elements after i move one place to the left. The comments `// 0<=i<=n` and `// 0<=i<n` in the code fragment are a short way of writing these preconditions.

What happens when a precondition is violated is also part of the contract. In this interface the function returns `false` without changing anything. Another design may leave the precondition to the caller's responsibility and perform no check. This is faster, but a wrong call leads to undefined behavior. Whichever way is chosen, it must be documented. Whether a function whose contract has not been written works correctly or incorrectly cannot even be discussed.

9.1.2 Opaque type: separating the interface from the implementation

The first line in the interface, `typedef struct list List;`, declares only the name of the structure, not its fields. After such a declaration, `struct list` is an **incomplete type**: the compiler recognizes its name but does not know its size. A variable cannot be defined of an incomplete type, but a pointer to it can. All pointers have the same size, and this is the only thing the compiler needs to know. All the functions in the interface therefore take `List *`, not `List`.

The fields of the structure are written only in the file `list.c`, which contains the implementation. A program that uses the header file cannot write an expression like `L->n`, because it does not know that a field called `n` exists. It can touch the list only through the functions in the interface. A type whose inside cannot be seen from outside is called an **opaque type**. `FILE` in the standard library is a familiar example: programs work with `FILE *` but do not use the fields of the structure.

The following program uses the list only through the interface. It first inserts five values at the front and then deletes the element at position 2:

```c
#include <stdio.h>
#include "list.h"                       // interface only

int main(void) {
    List *L = list_create();
    int x, k;
    if (L == NULL) return 1;
    for (k = 1; k <= 5; k++)
        list_insert(L, 0, 10 * k);      // always at front: 50 40 30 20 10
    list_delete(L, 2, &x);              // removes 30
    printf("%d %d %d\n", list_length(L), x, list_find(L, 20));
    list_destroy(L);
    return 0;
}
```

Because every insertion is at the front, after the loop the list is *(50, 40, 30, 20, 10)*. `list_delete(L, 2, &x)` removes 30 and the list becomes *(50, 40, 20, 10)*. The position of 20 is now 2. The program prints `4 30 2`. This program was compiled and run with two separate `list.c` files, one using an array and the other a linked list; both implementations appear in these notes. In both cases the output is the same and not a single line of `main` has changed. This is the gain of the opaque type. When the implementation changes, only `list.c` is recompiled; the hundreds of files that use the list stay as they are.

Common mistake

Writing `List L;` or `malloc(sizeof(List))` in the using file. `List` is an incomplete type in that file and the compiler does not know its size; both lines give a compile error. Only `list_create`, which knows the size, can allocate the list. This rule is not a restriction but the very purpose of the opaque type.

9.2 Implementation with an array

The most direct way to implement the list ADT is to keep the elements in order in an array. a_i is written into the i-th cell of the array. This layout is called **sequential allocation**: the logical order and the order in memory are the same.

9.2.1 Sequential layout and deletion

The first implementation in the slides uses a fixed-size array and an item counter:

```c
#define CAPACITY 100
typedef struct {
    int a[CAPACITY];
    int n;                  // item count
} ArrayList;

bool array_delete(ArrayList *L, int i, int *x) {
    int k;
    if (i < 0 || i >= L->n) return false;
    *x = L->a[i];
    for (k = i; k < L->n - 1; k++)
        L->a[k] = L->a[k + 1];      // shift left
    L->n--;
    return true;
}
```

`ArrayList` has two fields. `a` is the array that can hold at most `CAPACITY` elements; `n` records how many cells are currently filled. The filled cells are always those from `a[0]` to `a[n-1]`; no gap is left between them. This rule is the **invariant** of the structure and must be preserved at the end of every operation. Since `int` is 4 bytes, the structure occupies 404 bytes. This size does not change whether the list holds one element or a hundred.

`array_delete` first checks the precondition. If i is not a valid position, it returns `false` without touching anything. If it is valid, it copies the value to be deleted into `*x`. Then it moves every element after i one place to the left: `a[i] = a[i+1]`, `a[i+1] = a[i+2]`, and so on up to `a[n-2] = a[n-1]`. This is why the loop condition is `k < L->n - 1`; if `k` reached $n - 1$, `a[n]` would be read, and there is no valid element in that cell. At the end `n` decreases by one. The old value stays in the last cell, but because it is now outside `n`, it is not considered part of the list.

Access to *a[i]* is a single address computation: $i \cdot sizeof(int)$ is added to the start address of the array. This computation is independent of the size of i; `list_get` is therefore $O(1)$. In deletion, the number of elements moved is $n - 1 - i$. For $i = n - 1$, that is, deleting from the end, no move is needed and the cost is $O(1)$. For $i = 0$, that is, deleting from the front, $n - 1$ moves are made and the cost is $\Theta(n)$. If all positions are equally likely to be deleted, the average number of moves is $(n - 1)/2$; the average case is also $\Theta(n)$.

9.2.2 Code trace: `array_insert(&L, 1, 9)`

Insertion is like the mirror image of deletion. Where deletion closes the gap, insertion first makes room:

```c
bool array_insert(ArrayList *L, int i, int x) {
    int k;
    if (L->n == CAPACITY || i < 0 || i > L->n)
        return false;          // full/invalid
    for (k = L->n; k > i; k--)
        L->a[k] = L->a[k - 1];     // shift right
    L->a[i] = x;
    L->n++;
    return true;
}
```

The function rejects in three cases: if the array is full, if i is negative, or if $i > n$. The `||` operator is evaluated from left to right with short-circuiting; at the first true condition the others are not looked at. Then the loop starts at `k = n` and moves backward. In each round it moves `a[k-1]` to `a[k]`. The loop stops when `k == i`. At this moment the old value of `a[i]` has been copied to `a[i+1]` and the cell is ready to receive the new value.

The direction of the move is critical. If the loop went from left to right, starting at `k = i`, the assignment `a[i+1] = a[i]` would destroy the old value of `a[i+1]`. In the next round, the value read from that cell would already be a copy of `a[i]`, and the whole right side would be filled with the same value. In a right-to-left move, every cell is copied to the right before it is overwritten.

Example 9.1

Let the list be *[4, 7, 2]*, $n = 3$. The value 9 is to be inserted at position 1 (positions start at 0). Table 9.1 shows the effect of each line and Figure 9.1 shows four moments of the array.

Line	`k`	Condition / action	`a[0..3]`	`n`
Entry	?	`i = 1`, `x = 9`	4, 7, 2, —	3
`if (…)`	?	not full, $0 \le 1 \le 3$: continue	4, 7, 2, —	3
`for`	3	$3 > 1$ true	4, 7, 2, —	3
`a[k] = a[k-1]`	3	`a[3] = a[2]` : 2 moved right	4, 7, 2, 2	3
`for`	2	$2 > 1$ true	4, 7, 2, 2	3
`a[k] = a[k-1]`	2	`a[2] = a[1]` : 7 moved right	4, 7, 7, 2	3
`for`	1	$1 > 1$ false: the loop ends	4, 7, 7, 2	3
`a[i] = x`	1	9 into the vacated position	4, 9, 7, 2	3
`n++`	—	list *[4, 9, 7, 2]*	4, 9, 7, 2	4

Table 9.1: Trace of `array_insert(&L, 1, 9)`. The array column shows `a[0..3]`; `a[3]` does not belong to the list at the start (—).

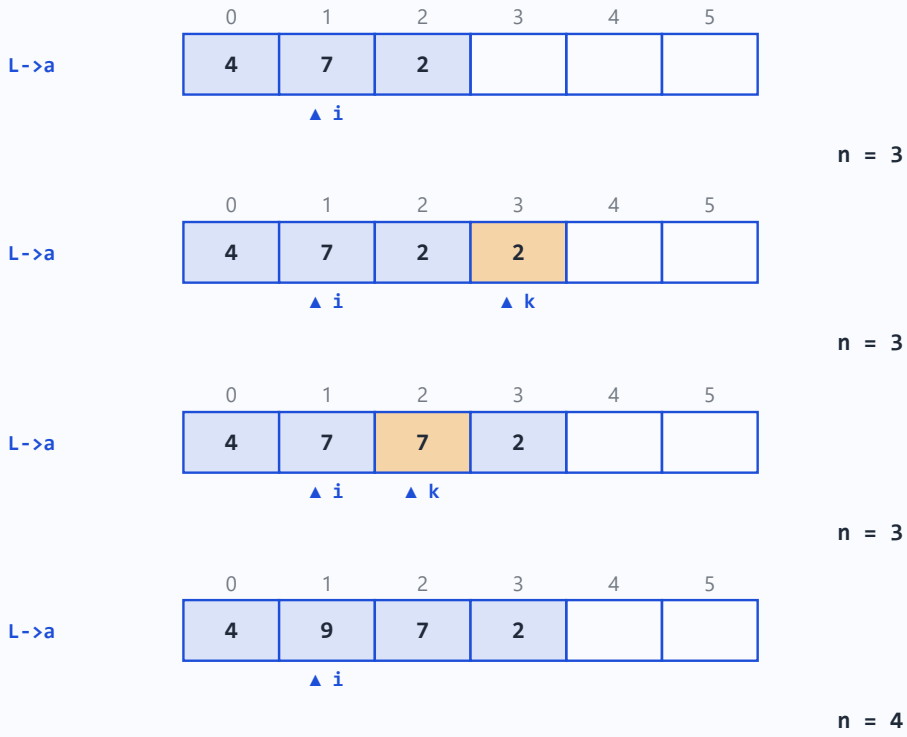


Figure 9.1: Four moments of the array during `array_insert(&L, 1, 9)`. From top to bottom: the start; 2 copied into `a[3]`; 7 copied into `a[2]` (at this moment 7 sits in two cells at once); 9 written into `a[1]` and `n` becoming 4.

Two intermediate states in the table deserve attention. After `a[3] = a[2]`, the value 2 is in two cells; after `a[2] = a[1]`, the same holds for 7. These temporary copies are not a bug. The value to be overwritten has always been moved to the cell on its right beforehand. At the end of the loop only `a[i]` carries an extra copy, and `x` is written over it right afterward.

The number of shifts is $n - i$. Insertion at the end ($i = n$) makes no shifts; insertion at the front ($i = 0$) shifts all n elements. The worst case is therefore $\Theta(n)$. An experiment that inserts at the front of an empty array a hundred times confirms this: the total number of shifts came out as $0 + 1 + \ldots + 99 = 4950$. That is, n insertions at the front amount to $n(n-1)/2$ shifts in total, $\Theta(n^2)$ work. The hundred-and-first insertion into the full array returns `false`.

9.2.3 Capacity and doubling

The fixed array has two drawbacks. The capacity is chosen at compile time and cannot be exceeded. If the chosen capacity is large, short lists also hold the same large space for nothing. The solution is to allocate the array in heap memory and grow it with `realloc` when needed. Such an array is called a **dynamic array**. The real question is by how much to grow it.

If the array is grown by one cell every time it fills up, every insertion calls `realloc`. If `realloc` cannot grow the block in place, it allocates a new block and copies all the old elements there. The number of elements copied over n insertions is, in the worst case, $1 + 2 + \ldots + (n-1)$, that is, $\Theta(n^2)$. If

the capacity is doubled each time, the picture changes. The `list.c` below implements the interface with this method. The capacity starts at 4 and doubles every time the array fills up:

```
// list.c: struct list is defined only here (opaque type)
struct list {
    int *a;                             // array on the heap
    int n, cap;                         // item count, capacity
};

List *list_create(void) {
    List *L = malloc(sizeof *L);
    if (L == NULL) return NULL;
    L->n = 0;
    L->cap = 4;
    L->a = malloc(L->cap * sizeof *L->a);
    if (L->a == NULL) { free(L); return NULL; }
    return L;
}

void list_destroy(List *L) {
    if (L) { free(L->a); free(L); }
}

int list_length(const List *L) { return L->n; }

bool list_get(const List *L, int i, int *x) {
    if (i < 0 || i >= L->n) return false;
    *x = L->a[i];
    return true;
}

bool list_insert(List *L, int i, int x) {
    int k;
    if (i < 0 || i > L->n) return false;
    if (L->n == L->cap) {               // full: double it
        int *bigger = realloc(L->a, 2 * L->cap * sizeof *L->a);
        if (bigger == NULL) return false; // old array stays intact
        L->a = bigger;
        L->cap *= 2;
    }
    for (k = L->n; k > i; k--)
        L->a[k] = L->a[k - 1];
    L->a[i] = x;
    L->n++;
    return true;
}

bool list_delete(List *L, int i, int *x) {
    int k;
    if (i < 0 || i >= L->n) return false;
    *x = L->a[i];
    for (k = i; k < L->n - 1; k++)
        L->a[k] = L->a[k + 1];
    L->n--;
    return true;
```

```
}

int list_find(const List *L, int x) {
    for (int k = 0; k < L->n; k++)
        if (L->a[k] == x) return k;
    return -1;
}
```

The structure now holds not the array itself but the address of an array on the heap. `list_create` allocates two separate blocks: one for the structure and one for the array. If the second allocation fails, the first is also returned; otherwise the structure would leak. `list_destroy` releases the blocks in reverse order. First the array and then the structure are returned, because the address of the array is inside the structure.

The `realloc` call in `list_insert` first puts its result into a separate variable. Had `L->a = realloc(L->a, …)` been written and memory run out, `realloc` would return `NULL` but would not release the old block. The moment `NULL` is written into `L->a`, the address of the old block is lost and the whole list leaks. The variable `bigger` keeps the old array safe in this case; the function returns `false` and the list stays intact.

When the example program above was run with this implementation, a single `realloc` call was seen at the fifth insertion: the capacity went from 4 to 8, and the block from 16 bytes to 32 bytes. The program made two `malloc` and two `free` calls; there is no leak.

Theorem 9.1 (Amortized cost of doubling)

Let n insertions at the end be made into a dynamic array that starts with capacity $c_0 \geq 1$ and doubles every time it fills up. The total number of elements copied during the growth steps is less than $2n$. Hence the **amortized cost** of insertion at the end is $O(1)$.

Proof. Growth happens when a new insertion arrives while the item count equals the capacity, and at that moment all the elements are copied. The capacities go $c_0, 2c_0, 4c_0, \ldots$. Suppose the last growth was made at capacity $c_0 2^k$; this growth copies $c_0 2^k$ elements, and at that moment the array holds that many elements. Hence $c_0 2^k < n$. The total number of copies is a geometric sum: $c_0(1 + 2 + \ldots + 2^k) = c_0(2^{k+1} - 1) < 2 \cdot c_0 2^k < 2n$. If there was no growth at all, the total is zero. The n insertions themselves make n writes. Together with the copies, the total work is less than $3n$ and the share per insertion is constant. ■

Amortized cost is the total cost of a sequence of operations divided by the number of operations. Not every single insertion is cheap. The insertion that coincides with a growth takes $\Theta(n)$; but such insertions are so rare that the total work stays linear. The growth factor may be another constant, such as 1.5 instead of 2; the proof stays the same, only the constant changes. Adding a fixed amount (for example 10 cells each time), however, is not enough. In that case the number of growth steps is $n/10$ and the total number of copies is again $\Theta(n^2)$.

Common mistake

Reading amortized $O(1)$ as "every insertion is $O(1)$". In a real-time system, where even a single insertion taking long is unacceptable, the worst case matters, not the average. Then a linked list or a preallocated array is chosen instead of a dynamic array. Moreover, this theorem describes only insertion at the end. Insertion at the front remains $\Theta(n)$ because of shifting, even without growth.

9.3 Linked implementation

In the linked implementation each element sits in a separate node, and each node carries the address of the next one. Insertion and deletion no longer move elements; they only change links. The node used in this section is the classic singly linked list node, made of an integer value and a link.

9.3.1 Insertion with a pointer to a pointer

```
typedef struct node *node_pointer;
struct node { int data; node_pointer link; };
typedef struct { node_pointer head; int n; } LinkedList;

bool ll_insert(LinkedList *L, int i, int x) {
    node_pointer *pp = &L->head, new_node;
    if (i < 0 || i > L->n) return false;
    while (i-- > 0)
        pp = &(*pp)->link;       // walk to link i
    new_node = malloc(sizeof *new_node);
    if (new_node == NULL) return false;
    new_node->data = x;
    new_node->link = *pp;
    *pp = new_node;              // head or middle: same
    L->n++;
    return true;
}
```

The `LinkedList` structure holds the head of the list and its length together. Because the length is stored in a separate field, `list_length` does not have to count the list and answers in $O(1)$. The price is that `n` must not be forgotten when updating in every insertion and deletion.

To insert at position i in a singly linked list, the link that will point to the new node has to be changed. If $i = 0$, the link to change is the head of the list, that is, `L->head`. If $i > 0$, the link to change is the `link` field of the $(i - 1)$-th node. In the classical formulation these two cases are handled separately: one branch is written for insertion at the head and another, which finds the previous node, for insertion in the middle.

`ll_insert` combines the two cases with a single idea. It holds not the link to be changed but the **address** of that link. `node_pointer *pp` is a variable that holds the address of a node pointer; it is therefore called a **pointer to pointer**. Initially `pp = &L->head`. In each round of the loop, `pp = &(*pp)->link` moves to the address of the next link. This expression is read from the inside out: `*pp` is the value of the current link, that is, the address of a node; `(*pp)->link` is the link field of that node, and `&(*pp)->link` is the address of this field. After *i* rounds, `pp` holds the address of the link that points to the node at position *i*.

The linking takes two lines. `new_node->link = *pp` links the new node to the node that currently stands at that position. `*pp = new_node` turns that link to the new node. If `pp` points to `&L->head`, the head of the list changes; if it points to the `link` field of a node, that node's link changes. The code does not distinguish the two cases, because both are "changing the value of a pointer". The order of the lines matters. Had `*pp = new_node` been written first, `new_node->link = *pp` would link the new node to itself and the rest of the list would be lost.

Line	`i`	`pp`	`*pp`	List
Entry	1	`&B.head`	&node(4)	4 → 7 → 2
`while (i-- > 0)`	0	*1 > 0* true; `i` became 0		
`pp = &(*pp)->link`	0	&node(4)->link	&node(7)	4 → 7 → 2
`while (i-- > 0)`	−1	*0 > 0* false; the loop ends		
`new_node = malloc(…)`, value	−1	&node(4)->link	&node(7)	new node: 9, link garbage
`new_node->link = *pp`	−1	&node(4)->link	&node(7)	9 → 7 (4 still points to 7)
`*pp = new_node`	−1	&node(4)->link	&node(9)	4 → 9 → 7 → 2
`L->n++`	`n = 4`; returns `true`			

Table 9.2: Trace of `ll_insert(&B, 1, 9)`; initially the list is 4 → 7 → 2, `n = 3`.

The `i-- > 0` in the loop condition first compares the old value of `i` with 0 and then decreases `i` by one. The loop therefore runs exactly *i* times and `i` is −1 on exit. Because the function does not use `i` afterward, this value causes no problem. Had the same call been made with *i = 0*, the loop would not have run at all, `pp` would have stayed at `&B.head`, and `*pp = new_node` would have changed the head of the list. Insertion at the head thus ceases to be a special case.

The cost has two parts. Walking to the position is *i* rounds, that is, *O(i)*. Linking is a constant number of assignments, that is, *O(1)*. The total is *O(i)*: insertion at the head is *O(1)* and insertion at the end is *O(n)*. In the array the opposite held. In a linked list no element moves and there is no capacity limit; the list grows as long as `malloc` provides memory. In the experiment, 7, 4, and 2 were inserted at positions 0, 0, and 2 in turn, and the list became 4 → 7 → 2. Then `ll_insert(&B, 1, 9)` gave 4 → 9 → 7 → 2, and invalid positions (6 and −1) were rejected with `false`. A single node is 16 bytes: a 4-byte `int`, 4 bytes of alignment padding, and an 8-byte pointer.

9.3.2 The same interface, another implementation

To see the power of the opaque type, a linked implementation of the same interface can be written. The structure changes only inside `list.c`:

```c
// linked version of list.c: list.h and the using code do not change
typedef struct node *node_pointer;
struct node { int data; node_pointer link; };
struct list { node_pointer head; int n; };

List *list_create(void) {
    List *L = malloc(sizeof *L);
    if (L) { L->head = NULL; L->n = 0; }
    return L;
}

void list_destroy(List *L) {
    node_pointer p, q;
    if (L == NULL) return;
    for (p = L->head; p; p = q) { q = p->link; free(p); }
    free(L);
}

int list_length(const List *L) { return L->n; }

static node_pointer *link_address(List *L, int i) {
    node_pointer *pp = &L->head;          // address of link i
    while (i-- > 0) pp = &(*pp)->link;
    return pp;
}

bool list_get(const List *L, int i, int *x) {
    node_pointer p = L->head;
    if (i < 0 || i >= L->n) return false;
    while (i-- > 0) p = p->link;
    *x = p->data;
    return true;
}

bool list_insert(List *L, int i, int x) {
    node_pointer *pp, new_node;
    if (i < 0 || i > L->n) return false;
    pp = link_address(L, i);
    new_node = malloc(sizeof *new_node);
    if (new_node == NULL) return false;
    new_node->data = x;
    new_node->link = *pp;
    *pp = new_node;
    L->n++;
    return true;
}

bool list_delete(List *L, int i, int *x) {
    node_pointer *pp, victim;
    if (i < 0 || i >= L->n) return false;
    pp = link_address(L, i);
    victim = *pp;
```

```
    *x = victim->data;
    *pp = victim->link;                 // skip the node
    free(victim);
    L->n--;
    return true;
}

int list_find(const List *L, int x) {
    int k = 0;
    for (node_pointer p = L->head; p; p = p->link, k++)
        if (p->data == x) return k;
    return -1;
}
```

The helper function `link_address` is the very walk in `ll_insert`. Because it is `static`, it is visible only from this file and is not part of the interface. `list_delete` uses the same walk for deletion. `*pp = victim->link` skips the node to be deleted, and then the node is returned. Deleting from the head and deleting from the middle are the same three lines. While walking over the nodes, `list_destroy` saves the address of the next one into `q` before `free`. Reading the `link` field of a node that has been freed is undefined behavior.

The example program was also compiled with this file and gave the same output, `4 30 2`. This time six `malloc` calls were made: five for nodes and one for the structure itself. All were returned with six `free` calls. For the using code, the only visible difference between the two implementations is speed.

9.3.3 Array or linked list?

The question of which of the two implementations is better has no single answer. The answer depends on which operations the program performs often. Table 9.3 gives the comparison from the slides.

Criterion	Array (sequential)	Singly linked list
Access to the *i*-th element	*O(1)*	*O(i)*
Insertion / deletion at the front	*O(n)* shifting	*O(1)*
Insertion at the end	*O(1)** (if there is room)	*O(n)*, *O(1)* with a tail pointer
Insertion at a known position	*O(n)*	*O(1)* (if the previous node is known)
Search (unsorted)	*O(n)*	*O(n)*
Search (sorted)	*O(log n)* binary search	*O(n)*
Memory	Data only; unused capacity	A pointer per node
Cache	Contiguous: very good	Scattered: poor

Table 9.3: Comparison of the sequential (array) and linked implementations.

The asterisk in the table reminds us that insertion at the end of a dynamic array is amortized *O(1)*. The row "insertion at a known position" carries a subtle difference. In an array, even if the position is known, insertion is *O(n)* because of shifting. In a linked list, if the node before the insertion point is at hand, two links change and the job is done. If one has to walk from the head to reach this node, the cost is again *O(i)*.

Binary search relies on being able to reach the middle element in *O(1)*. In a linked list, even going to the middle takes *O(n)*, so binary search on a sorted linked list gains nothing; it takes as long as linear search.

The last two rows describe differences that asymptotic notation does not capture. In terms of memory, an array holds only the data for each element, but the unfilled part of the capacity is wasted. A linked list holds no empty cells, but each node carries a pointer. In a list of `int` this means a 16-byte node for 4 bytes of data. In terms of the processor the difference is even larger. The processor brings memory into the **cache** not byte by byte but in blocks of 64 bytes. When a cell of an array is read, its neighbors come along as well, and subsequent accesses are very fast. The nodes of a linked list may be scattered in memory; every `p = p->link` may require a new block to be fetched. The same *O(n)* traversal can be several times faster on an array than on a linked list.

The rule summarized in the slides is this: if frequent access or binary search is needed, use an array; if frequent insertion and deletion in the middle is done, or if the size cannot be predicted in advance, use a linked list. An opaque interface also makes it possible to postpone this decision. The program is first written with one implementation and measured; if necessary, only `list.c` is changed.

9.4 Generalized lists

In a linear list each element is a single value: a number, a letter, or a record. Much data, however, is nested. In a file system, a directory contains subdirectories; in a menu, submenus; in a mathematical expression, parenthesized subexpressions. To hold such data naturally, it is enough to allow the elements of a list to be lists as well.

9.4.1 Definition, head, and tail

Definition 9.2

Let $A = (\alpha_0, \alpha_1, \ldots, \alpha_{n-1})$ be a finite sequence of elements. Each α_i is either an **atom** (a single indivisible value) or itself a list of this form. Such an A is called a **generalized list**. The elements that are lists are called the **sublists** of A. A is the name of the list and n is its length. If $n \geq 1$, then α_0 is the **head** of the list and $(\alpha_1, \ldots, \alpha_{n-1})$ is its **tail**.

The definition refers to itself: a generalized list is a list whose elements can be generalized lists. Definitions of this kind are called recursive definitions. It is therefore natural that the routines that process such a structure are recursive too. A linear list is a special generalized list all of whose elements are atoms.

Attention must be paid to the asymmetry between head and tail. The head is an element; it can be an atom or a list. The tail is always a list, if necessary the empty list. The head of the list *(a)* is the atom *a* and its tail is *()*. These two operations are the foundation of the LISP language designed by John McCarthy in 1958. In LISP the head is called `car` and the tail `cdr`. LISP programs are also written in the same way, as nested lists: the expression `(+ 1 (* 2 3))` is both a program and a generalized list of three elements.

List	Length	Description
D = ()	0	empty list
A = (a, (b, c))	2	the first element is an atom and the second is the sublist *(b, c)*; head(*A*) = *a*, tail(*A*) = *((b, c))*
B = (A, A, ())	3	shared sublist: the first two elements are the same *A*; the third is the empty list
C = (a, C)	2	recursive list: it contains itself, *C = (a, (a, (a, ...)))*

Table 9.4: Examples of generalized lists.

Every row in the table carries its own lesson. The length of *A* is 2, not 3. The length counts only the elements at the top level; *(b, c)* is a single element. The tail of *A* is *((b, c))*, not *(b, c)*. The tail is the *remaining list* after the head is removed, and this list has a single element; that single element is the list *(b, c)*.

Two of the three elements of *B* are the same list *A*. This does not mean two copies of *A*. There is a single *A* in memory, and two elements of *B* point to it. Such lists are called **shared** lists. The third element of *B* is the empty list. The empty list is also a list; it increases the length of *B* by one.

C contains itself as an element. If one tries to expand it on paper, it goes on forever as *(a, (a, (a, ...)))*. In memory, however, it is a finite structure: the second element points to the head of the list itself. Such lists are called **recursive** lists. Sharing and recursion are the power of generalized lists; but most of the routines written in these notes are correct for lists that contain neither. The limits are stated separately next to each routine.

9.4.2 Example: a multivariable polynomial

A single-variable polynomial is easily held in a linked list in which each node carries a coefficient and an exponent. With polynomials that contain several variables, the job gets harder. A fixed structure that holds the exponent of every variable in each term grows as the number of variables grows, and most of its fields stay empty. A generalized list instead groups the polynomial in nested levels.

The example in the slides is the following polynomial:

$$P(x, y, z) = x^{10}y^3z^2 + 2x^8y^3z^2 + 3x^8y^2z^2 + x^4y^4z + 6x^3y^4z + 2yz$$

The polynomial is first grouped by powers of *z*. There are three terms containing z^2 and three terms containing *z*:

$$P = (x^{10}y^3 + 2x^8y^3 + 3x^8y^2)\, z^2 + (x^4y^4 + 6x^3y^4 + 2y)\, z$$

The coefficients inside the parentheses are now polynomials in *x* and *y* only. These are grouped in turn by *y*:

$$P = ((x^{10} + 2x^8)\, y^3 + 3x^8\, y^2)\, z^2 + ((x^4 + 6x^3)\, y^4 + 2y)\, z$$

The innermost coefficients contain only *x* and are ordinary polynomials with numeric coefficients. Every level has the same form: a variable and coefficients multiplied by powers of that variable. A coefficient is either an integer or a polynomial of a lower level. This fits the definition of the generalized list exactly. The slides hold each element with the following structure:

```
typedef enum { VAR, PTR, NO } tag_t;
typedef struct pnode *poly_pointer;
struct pnode {
    tag_t tag;              // three kinds
    union {
        char var;           // VAR: variable name
        poly_pointer dlink; // PTR: coef is a list
        int coef;           // NO : integer coef
    };
    int exp;                // exponent
    poly_pointer link;
};
```

The `tag` field says what kind of element this is and takes three values. A `VAR` element is the first element of a level and carries the variable name of that level in `var`; its exponent is 0 and unused. A `PTR` element is a term whose coefficient is a sublist; `dlink` points to that sublist. A `NO` element is also a term, but its coefficient is an integer held in `coef`. In both term kinds, `exp` is the exponent of the variable of the level. `link` points to the next element on the same level.

The three pieces of information are never needed at the same time: an element carries either a variable name, or a sublist, or an integer. The three fields are therefore put into a `union`. All the fields of a `union` start at the same memory address; the size of the union is that of the largest field. Which field is valid is known from `tag`. This arrangement, in which a union is used together with a field that indicates its kind, is called a **tagged union**. If the tag is read wrongly, for example if the `dlink` field of a `NO` element is followed, an integer is mistaken for an address and the program crashes. Checking the tag before every access is the programmer's responsibility.

Figure 9.2 shows the memory layout of two examples. The upper drawing is $3x^2y$. This term reads as $(3x^2) \cdot y^1$. The outer level is the level of y and has a single term: a `PTR` element with exponent 1 whose coefficient is a sublist. The sublist is the level of x, and its only term is a `NO` element with coefficient 3 and exponent 2.

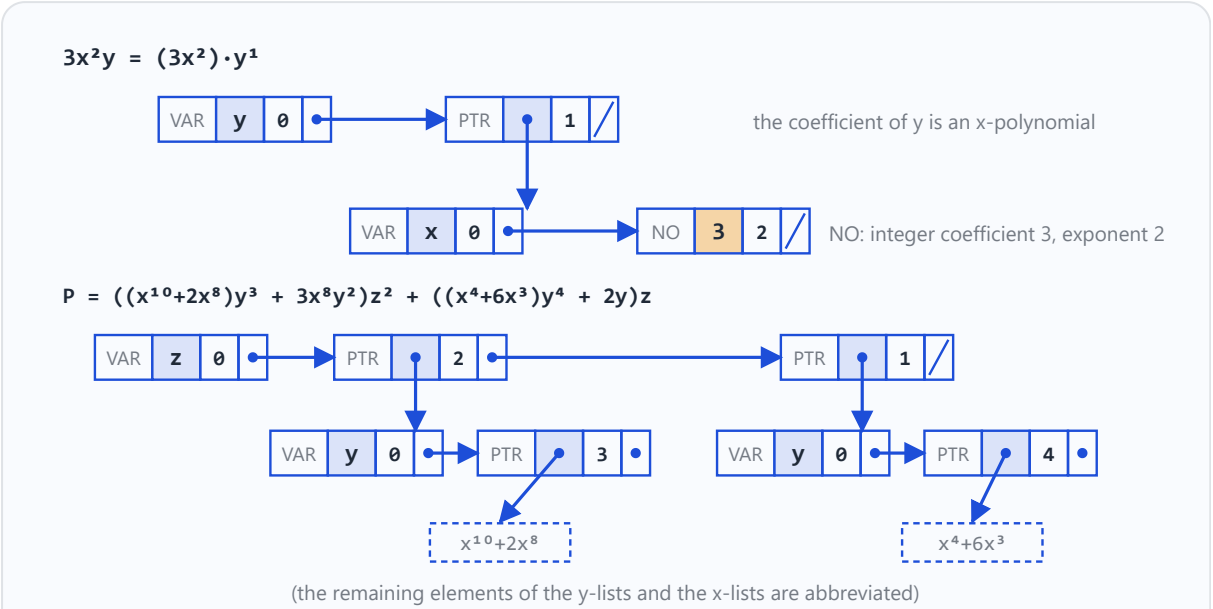


Figure 9.2: Generalized-list representation of multivariable polynomials. Top: $3x^2y$: the single term of the `VAR y` level is a `PTR` whose coefficient is the sublist starting with `VAR x`; there stands `NO 3 2`, that is, $3x^2$. Bottom: the top two levels of $P(x, y, z)$: at the z level, two terms with exponents 2 and 1, each of whose coefficients is a y-polynomial. The innermost x-polynomials are abbreviated with dashed boxes.

In the lower drawing the z level has three elements: `VAR z` and two `PTR`s. The first `PTR` has exponent 2 and its coefficient is the polynomial $(x^{10} + 2x^8)\,y^3 + 3x^8\,y^2$. The list of this polynomial starts with `VAR y` and contains two `PTR`s with exponents 3 and 2. To save space, the drawing shows only the first term of the y-lists and shows the x-polynomials abbreviated. The coefficient of the term $2y$ in the second y-list is just the number 2; this term needs no sublist and a `NO` element suffices. The whole structure is thus 17 elements: 3 at the z level, 3 in each of the two y lists, and 8 in total in the three x lists. On x86-64 each element is 32 bytes: after each of the four-byte `tag` and `exp` fields, four bytes of padding remain to align the 8-byte pointers.

The routine that computes the value of such a structure is as orderly as the structure itself. At each level, the coefficient is either read directly or computed recursively as the value of the sublist; then it is multiplied by the power of the variable and added up:

```c
// P: list starting with a VAR element; d['x'-'x'], d['y'-'x'], d['z'-'x']: variable values
long poly_value(poly_pointer P, const long d[]) {
    long sum = 0, v = d[P->var - 'x'];  // value of this level's variable
    long coef, pw;
    int k;
    for (poly_pointer p = P->link; p; p = p->link) {
        coef = (p->tag == PTR) ? poly_value(p->dlink, d)   // coef is a list
                               : p->coef;                  // NO: integer
        for (pw = 1, k = 0; k < p->exp; k++) pw *= v;
        sum += coef * pw;
    }
    return sum;
}
```

The function reads the name of the variable from the `VAR` element and takes its value from the array `d`; `'x'` is in cell 0, `'y'` in cell 1, and `'z'` in cell 2. Then it walks over the terms along `link`. In a `PTR` term the coefficient is the value of the sublist at the same point. For example, for $x = 2, y = 3, z = 1$, the outermost level adds up the z^2 and z terms; because $z = 1$, the result is the sum of the values of the two y-polynomials. The first y-polynomial gives $(2^{10} + 2 \cdot 2^8) \cdot 27 + 3 \cdot 2^8 \cdot 9 = 1536 \cdot 27 + 6912 = 48384$ and the second $(16 + 48) \cdot 81 + 2 \cdot 3 = 5190$. The total is 53574. This value is the same as that of the expanded polynomial. The list was also compared with the explicit formula at all 343 points in the range $-3 \le x, y, z \le 3$, and no difference was found.

This representation is not the only option. The same polynomial can be stored by choosing a different variable order (grouping by x first), or with a flat list that holds each term with an exponent vector. What matters is that conversion routines can also be written for the chosen representation: from text to structure, from structure to text, from one representation to another. A representation counts as a solution only together with these routines.

9.4.3 Element structure and memory layout

The polynomial example uses three kinds of elements. In a general-purpose generalized list, two kinds are enough: atom and sublist. All the routines in the rest of these notes use the following structure:

```c
typedef enum { FALSE, TRUE } boolean;
typedef struct list_node *list_pointer;
struct list_node {
    boolean tag;          // FALSE: atom
    union {
        char data;        // atom value
        list_pointer dlink; // sublist head
    };
    list_pointer link;    // next on this level
};
```

The `tag` field is the tag. If it is `FALSE`, the element is an atom and its value is in `data`. If it is `TRUE`, the element is a sublist and `dlink` points to the first element of the sublist. In both cases `link` points to the next element on the same level. In short, an element has the form *[tag | data/dlink | link]*. In the drawings an atom element is marked with the letter F and a sublist element with the letter T.

Because the structure uses its own type, the pointer type `list_pointer` is named first; `struct list_node` is still an incomplete type at that moment, but defining a pointer to it is allowed. The type `boolean` is a two-valued enumeration (`enum`) defined by the slides themselves: `FALSE` takes the value 0 and `TRUE` the value 1. It must not be confused with `bool` from `<stdbool.h>`, available since C99. The two can be used side by side in the same program, because their names differ: `true` is lowercase and `TRUE` is uppercase.

The `union` has not been given a name. An **anonymous union** entered the standard with C11, and its fields are accessed directly as `L->data` and `L->dlink`. With a C99 compiler the union must be given a name, for example `u`; the access is then written `L->u.data` and `L->u.dlink`. All the code in these notes compiled without warnings with `-std=c11`.

The union saves memory. A `char` is 1 byte and a pointer is 8 bytes, and the union takes 8 bytes. The tag is 4 bytes, the union 8, and the link 8; because pointers must sit at addresses that are multiples of 8, 4 bytes of padding remain after the tag. On x86-64, `sizeof(struct list_node)` is 24 bytes. Without the union, that is, if `data` and `dlink` were separate fields, the element would grow to 32 bytes.

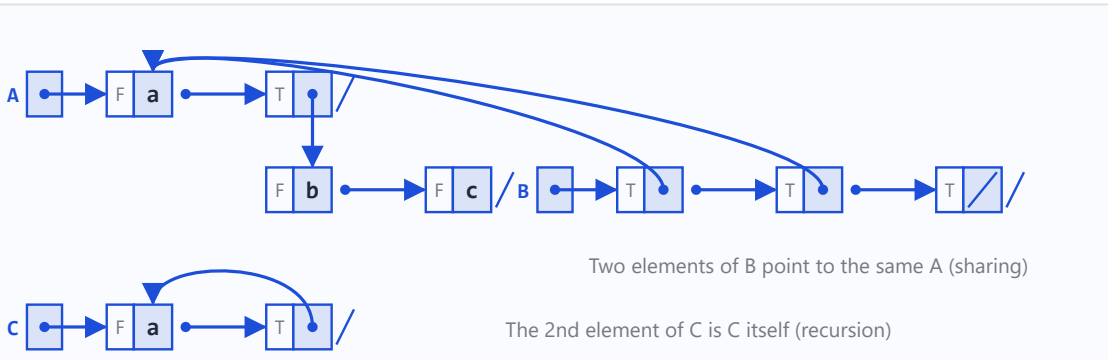


Figure 9.3: Memory layout of *A = (a, (b, c))*, *B = (A, A, ())*, and *C = (a, C)*. Each box is an element: the left cell is the tag (F atom, T sublist), the middle cell is the value or the sublist link, and the right cell is the link to the next element on the same level. The first two elements of *B* point to the same *A*; the sublist link of the third is null. The second element of *C* points to the head of *C* itself.

Figure 9.3 shows these three lists in memory. *A* has two elements: the atom *a* and a T element. The arrow from the middle cell of the T element descends to the first element of the sublist *(b, c)*. *b* and *c* are on the same row, linked to each other by `link`. Elements on the same level are always followed along one row; descending into a sublist means going one row down. This drawing convention makes it much easier to read nested structures.

B has three T elements. The arrows of the first two go to the same place, the first element of *A*. In the middle cell of the third there is a slash, that is, `NULL`: this element is an empty sublist. The difference between the empty list and "no element" shows here. The element *()* is an existing T element and increases the length of *B* by one; only the list it points to is empty. In *C* the arrow of the second element turns back and reaches the first element of the list. Any traversal that follows this arrow goes around forever between the same two elements.

> **Common mistake**
>
> Reading a field of the union without checking the tag. Writing `if (L->dlink)` may come out true for an atom element too: the character `'a'` written into the `data` field makes the first byte of the union 97, and the 8-byte value containing this byte is nonzero. `L->tag` must be examined first, and only the field of the union that is valid must be read.

9.4.4 Why generalized lists?

Three benefits of the generalized list stand out. The first is space. The whole structure of a tree, for example *A(B(E(K, L), F), C(G), D(H(M), I, J))*, can be written as a one-line text. This text carries all the information needed to rebuild the structure. A picture of the tree can be stored, or its image in memory can be dumped to disk as it is. Both work, but both take much more space than the text. The text is a kind of compression. A computer engineer should see this difference even when storing so small a piece of data.

The second is naturalness. Directories in a file system, drop-down menus, the nested elements of XML and JSON documents, and LISP expressions are hierarchical to begin with. The array `[1, [2, 3], []]` in a JSON document is itself the generalized list *(1, (2, 3), ())*.

The third concerns algorithms. When data is moved to a well-known structure, all the routines written for that structure become usable. This idea is common in engineering. In signal processing, a signal is moved to another space by a Fourier or Z transform, the processing is done there, and the result is converted back. Here too, data such as a tree is moved into a list; list routines such as copying, comparison, traversal, and printing are then ready.

9.4.5 Building a list from its linear form

Building a generalized list by hand, allocating it element by element with `malloc`, is both long and error-prone. A text such as *(a, (b, c))* is called the **linear form** of the list. A routine that reads the linear form and builds the list both lets programs take their data from outside and makes it easier to try out routines. All the example lists followed in these notes were built with this routine. The routine uses the following helper, which allocates an element for each item:

```
list_pointer get_node(void) {
    list_pointer p = malloc(sizeof *p);
    if (p == NULL) {                    // out of memory
        fprintf(stderr, "get_node: out of memory\n");
        exit(EXIT_FAILURE);
    }
    return p;                           // fields not initialized
}
```

`get_node` is a wrapper around `malloc(sizeof *p)` with a `NULL` check. The notation `sizeof *p` gives the size of the type that `p` points to; it leaves no need to write the type name again. If no memory is left, the program stops with an error message. This is a crude but honest choice: instead of doing a `NULL` check in each routine, the error is handled in one place. The fields of the returned element are not initialized; the caller must fill in every field.

The text is read character by character. The grammar is simple: a list begins with `(`, its elements are separated by commas, and it ends with `)`. Each element is either a single letter (an atom) or again a list. Because the grammar is recursive, the reader is recursive too:

```
// called while *s points at '('; once the list is built,
// *s points just after the matching ')'.
list_pointer parse(const char **s) {
    list_pointer head = NULL, *tail = &head, q;
    (*s)++;                             // skip '('
    while (**s != ')') {
        q = get_node();
        if (**s == '(') {               // sublist
            q->tag = TRUE;
            q->dlink = parse(s);
        } else {                        // atom
            q->tag = FALSE;
            q->data = **s;
            (*s)++;
        }
        q->link = NULL;
        *tail = q;                      // append at the end
        tail = &q->link;
        if (**s == ',') (*s)++;         // skip the separator
    }
    (*s)++;                             // skip ')'
    return head;
}
```

The function takes the position in the text as `const char **s`. The address of the position is passed because, after a sublist has been read, the caller must continue reading from where the sublist ended. The inner call advances `*s`, and this advance is visible from outside as well. This is another use of the pointer-to-pointer idea in `ll_insert`.

The loop runs until `)` is seen. In each round an element is allocated. If the next character is `(`, the element is a sublist and its `dlink` field is filled with the list built by the recursive call. Otherwise the character is an atom. The element is appended at the end. For this, the variable `tail` holds the address of the `link` field of the last element, or the address of `head` in an empty list. `*tail = q` fills `head` at the first element and the link of the previous element at later ones. Again two cases come down to one line. If a comma follows the element, it is skipped. On leaving the loop, the closing parenthesis is skipped and the list is returned.

The reader does not check for malformed text. An unclosed parenthesis leads to reading past the `'\0'` character. In a real program, every character should be checked against the expected one, and on an error the elements allocated so far should be released. Throughout these notes the reader was used only with correct texts. It was also tried together with the routine `print_linear`, which converts a list back to text, on 20,000 random texts: every text, read and written back, gave the same text.

Example 9.2

Let `"(a,(b,c))"` be read. The outer call skips `(` and sees `a`: an atom element is built, `head` is linked to it, and the comma is skipped. The next character is `(`. A new element is allocated, `tag = TRUE` is set, and the inner call begins. The inner call builds the atoms `b` and `c`, stops on seeing `)`, skips this parenthesis, and returns the first element of *(b, c)*. The outer call writes this address into the `dlink` of the T element and links the element behind *a*. Next comes the `)` of the outer list, and the outer loop ends too. Four elements have been allocated and the *A* of Figure 9.3 has been built.

9.5 Basic recursive routines

Most routines that work on a generalized list follow the same skeleton. When an element is reached, its kind is examined first. If the element is an atom, it is processed: copied, compared, or printed. If the element is a sublist, the same routine is called for that sublist. Then, with `link`, the routine moves on to the next element on the same level; this step is done either with a loop or, again, with a recursive call. In short: *if an atom, process it; if a sublist, descend; then continue*. The four routines in this section are four different uses of this pattern.

The way to write and read a recursive routine is to look at each call as a small contract. The contract has the form "if this call is given this list, it returns this". While writing the routine, the inner calls are assumed to obey this contract, and only the work of this call itself is considered. This assumption is the programming counterpart of mathematical induction. If the base case (the empty list) is handled correctly and every call is made with a smaller list, the routine is correct for all finite lists.

9.5.1 Copying

The first question in the slides asks for a routine that makes a copy of a list that has no sharing and no recursion. The copy must be a list of the same structure but made entirely of new elements: a change made in the copy must not affect the original list.

```
list_pointer copy(list_pointer L) {
    list_pointer q = NULL;
    if (L) {                    // L not NULL?
        q = get_node();         // get a node
        q->tag = L->tag;
        if (!L->tag) q->data = L->data; // atom
        else q->dlink = copy(L->dlink);
        q->link = copy(L->link);
    }
    return q;
}
```

The contract of the routine is this: `copy(L)` returns the first element of an independent copy of the list that `L` points to, that is, of the element `L` and all the elements reachable from it by `link`. For the empty list it returns `NULL`.

In the first line, `q` is initialized with `NULL`. If `L` is empty, the `if` block is skipped and this `NULL` is returned: the copy of the empty list is the empty list. That is the base case. If `L` is not empty, a new element is obtained with `get_node` and the tag is copied. In an atom element the value is copied as it is. In a sublist element, however, `dlink` is not copied; had it been copied, the new element would point to the old sublist and the copy and the original list would share the same sublist. Instead, a copy of the sublist itself is requested and the new element points to it. The last line copies the rest of the list, that is, the list starting at `L->link`, and links the new element to this copy.

The routine has two recursive calls, and both are made with a smaller list. The sublist in `L->dlink` and the remaining list starting at `L->link` contain only some of the elements of `L`. The calls therefore reach the empty list in the end and stop. This reasoning rests on the assumption that the list contains no recursion. In a list such as *C = (a, C)*, the sublist is the list itself and does not shrink.

At the moment a new element is allocated, its fields hold garbage values. The code always fills every field: `tag` at once, `data` or `dlink` according to the kind, and `link` last. A value can be written into the `link` field only when the copy of the rest of the list is complete. Elements are *allocated* from front to back, while links are *updated* from back to front. This order is seen in all the recursive routines in this section. A student who finds recursive thinking hard may propose doing the same job with a queue. It can be done, but the code gets longer and extra bookkeeping is needed to link the copies of the sublists to the right place. The recursive version translates the definition of the structure directly into code.

9.5.2 Code trace: `copy(L)`

Let *L = (a, (b, c))* be copied. The 20-step trace in the slides is collected into nine rows in Table 9.5. The stack column shows the call stack, with the call that is currently running at the far right. *k(x)* means the copy of element *x*.

Step	Stack (the one on the right is running)	Action
1	copy(a)	`get_node`: *k(a)* allocated; `tag = FALSE`, `data = 'a'`. The call `q->link = copy(L->link)` begins.
2	copy(a) · copy((b,c))	*k((b, c))* allocated; `tag = TRUE`. The sublist first: `q->dlink = copy(L->dlink)`.
3	… · copy((b,c)) · copy(b)	*k(b)* allocated; `data = 'b'`. Call for the rest.
4	… · copy(b) · copy(c)	*k(c)* allocated; `data = 'c'`. Call for the rest.
5	… · copy(c) · copy(NULL)	`L == NULL`: returns `NULL` without allocating anything. *k(c)* `->link = NULL`; `copy(c)` returns *k(c)*.
6	… · copy((b,c)) · copy(b)	*k(b)* `->link` ← *k(c)*; *k(b)* returns.
7	copy(a) · copy((b,c))	*k((b, c))* `->dlink` ← *k(b)*. Now the call for the rest.
8	… · copy((b,c)) · copy(NULL)	`NULL` is returned; *k((b, c))* `->link = NULL`; *k((b, c))* returns.
9	copy(a)	*k(a)* `->link` ← *k((b, c))*; *k(a)* returns. The copy is complete.

Table 9.5: Trace of `copy(L)`, *L = (a, (b, c))*.

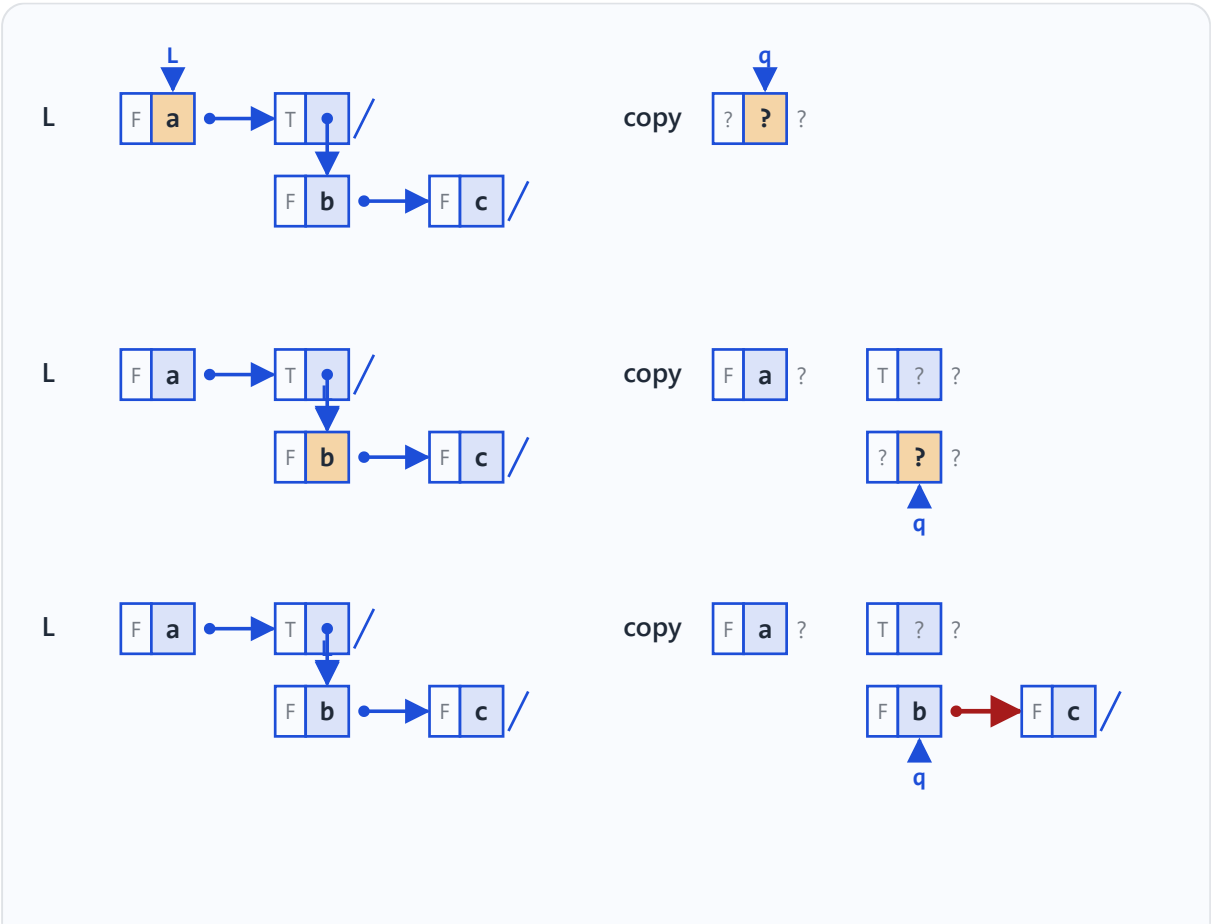




Figure 9.4: The original list (left) and the copy (right) during `copy(L)`. From top to bottom: the copy of *a* has been allocated, its link still garbage (?); the copy of *b* has been allocated, and the sublist link of the T element above it is still empty; on return, the link of *k(b)* has been written to *k(c)*; the copy is complete.

The most important observation of the trace is that the call stack goes at most four calls deep. The path is this: from *a* to *(b, c)* by `link`, from there to *b* by `dlink`, from there to *c* again by `link`, and then the empty list arrives. The depth of the stack depends not only on nesting but also on the number of elements on the same level, because moving along `link` is also a call. This detail matters for long lists.

By counting these calls we find the cost of the routine. For *(a, (b, c))*, `copy` is called six times: once for each of the four elements and twice for `NULL`. This number is no coincidence.

Theorem 9.2 (Number of calls)

Let a generalized list with no sharing and no recursion contain n elements in total, of which s are sublist elements. `copy` is called exactly $1 + n + s$ times for this list. Of the calls, n are made with an element and $1 + s$ with `NULL`. The running time of the routine is therefore $\Theta(n)$.

Proof. Every element becomes the parameter of a call exactly once. The first element of the top level comes in through the call from outside; the other elements come through the call `copy(L->link)` of the element before them, or through the call `copy(L->dlink)` of the T element that owns the sublist they are in. Because there is no sharing, no element is reached by two routes. Every call made with an element makes one call for `link`; T elements also make one call for `dlink`. So in addition to the call from outside, $n + s$ more calls arise, and the total is $1 + n + s$. Since n of them go to elements, the remaining $1 + s$ calls were made with `NULL`. Every call does a constant amount of work apart from its inner calls. Since $s \le n$, the total work is $\Theta(n)$. ■

The theorem was also confirmed by an experiment. With a copy instrumented with a counter, 8 calls were counted for *(a, (b, (c)))* ($n = 5$, $s = 2$), 12 for *((a, b), ((c, d), e))* ($n = 8$, $s = 3$), and 1 for the empty list. Copying also calls `malloc` n times and allocates $\Theta(n)$ memory for the copy itself. The extra memory needed for the stack is the length of the deepest call chain.

9.5.3 The limits of copying and freeing a list

The test list suggested by the slides is *L = ((a, b), ((c, d), e))*. This list has three levels, its first element is not an atom, and the first element of a sublist is itself a sublist. It therefore catches bugs that do not show up in simple examples. Figure 9.5 shows the memory layout of the list. The experiment showed that 8 `malloc` calls are made for this list, that the copy equals the original list, and that no element is common to the two lists.

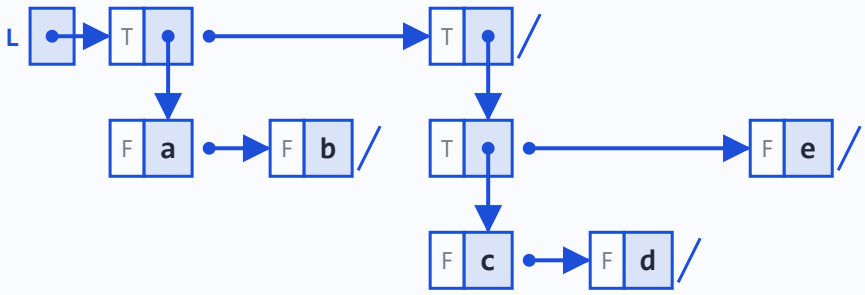


Figure 9.5: The test list *L = ((a, b), ((c, d), e))*. At the top level there are two T elements. The sublist of the second T again begins with a T element; *(c, d)* is at the lowest level.

The routine has two limits. The first is sharing. When *B = (A, A, ())* is copied, `copy` does not know that the first two elements of *B* point to the same list. A separate call `copy(L->dlink)` is made for every T element, and *A* is copied twice. With *A = (a, (b, c))*, the experiment counted 11 `malloc` calls: three elements for *B* and 4 each for the two copies of *A*. The value of the copy equals the original list, but its structure is different. In the original list the first two elements point to the same list, whereas in the copy they point to two separate lists. A change made to the first copy of *A* in the copy is not reflected in the second. The sharing has been lost. A copy that preserves sharing keeps the sublists already copied in a table; such a version is among the exercise solutions.

The second limit is recursion. For *C = (a, C)*, `copy(C)` first copies *a*, then asks for the copy of the sublist for the second element. The sublist is *C* itself; the same call starts over from the beginning. The calls never end. Because every call occupies space on the stack, after a while the call stack overflows and the operating system stops the program. This situation is called **stack overflow**. In an experiment on Windows, the program terminated with the code `0xC00000FD` (STATUS_STACK_OVERFLOW).

A copied list must be freed when its job is done. The routine that frees a list with no sharing and no recursion is the reverse of copying:

```
// Frees all elements of a list with no sharing and no recursion.
void free_list(list_pointer L) {
    list_pointer q;
    while (L) {
        if (L->tag)
            free_list(L->dlink);        // sublist first
        q = L;
        L = L->link;                    // link is read before free
        free(q);
    }
}
```

The routine walks the same level with a loop; it calls itself only for sublists. At every element the sublist is freed first. The address of the next element is read before `free`; the `link` field of a freed element can no longer be read. All the experiments in these notes were cleaned up with this routine, and the absence of leaks was checked with counting wrappers for `malloc` and `free`. In the main verification program, 216,150 `malloc` calls and the same number of `free` calls were counted.

Common mistake

Freeing a shared list with `free_list`. When *B = (A, A, ())* is freed, *A* is first released for the first T element. The second T element hands the same addresses to `free` once more. Releasing the same block twice is undefined behavior and corrupts the heap. The way to free shared lists safely is reference counting.

9.5.4 Equality

When are two lists equal? When their structures are the same and the corresponding atoms carry the same values. The structure being the same means that the two lists have the same length and that at every position both have either an atom or a sublist. For sublists the same definition is applied recursively. The question in the slides asks for this definition to be turned into code for lists with no sharing and no recursion:

```
boolean equal(list_pointer T, list_pointer L) {
    boolean x = TRUE;
    if (!T && !L) return TRUE;
    if (T && L && T->tag == L->tag) {
        if (!T->tag)             // atoms?
            x = (T->data == L->data) ? TRUE : FALSE;
        else                     // sublists
            x = equal(T->dlink, L->dlink);
        if (x == TRUE) return equal(T->link, L->link);
    }
    return FALSE;
}
```

The first condition is the base case: if both lists are empty, they are equal. The second condition checks that both elements exist and that their tags are the same. If one of them is empty, or if one is an atom and the other a sublist, this condition is false, the block is skipped, and the function returns `FALSE`. Inside the block the elements are compared first. For atoms the `data` fields are compared; for sublists the sublists themselves are compared recursively. The result is in `x`. If `x` is true, the answer depends on the equality of the remaining lists, and the result of that call is returned as it is. If `x` is false, the block ends and `FALSE` is returned again.

The code works correctly but is not tidy. The variable `x` is unnecessary; the form `(T->data == L->data) ? TRUE : FALSE` is long, considering that the comparison already gives 0 or 1. Three separate `return` flows make it hard to follow. A better version can be written. There is also a restriction that holds for both versions: two recursive lists such as *C = (a, C)* cannot be compared. When the sublists are compared at the second element, the same call starts over and the stack overflows, as in copying.

9.5.5 Code trace: `equal(T, L)`

Let *T = (a, (b, c))* and *L = (a, (b, d))* be compared. The difference is in the last atom, at the deepest level. Table 9.6 shows how the comparison descends and carries the result back up.

Call	Compared	Action	Result
1	*a* / *a*	both exist, both atoms; `'a' == 'a'`, `x = TRUE`; call 2 for the rest	result of call 2
2	*(b, c)* / *(b, d)*	both T; sublists first: call 3	`x` = result of call 3
3	*b* / *b*	atoms; `'b' == 'b'`, `x = TRUE`; call 4 for the rest	result of call 4
4	*c* / *d*	atoms; `'c' == 'd'` false, `x = FALSE`; the rest is not examined	`FALSE`
3	the `FALSE` from call 4 is returned unchanged		`FALSE`
2	`x = FALSE`; the block ends, `return FALSE`		`FALSE`
1	the `FALSE` from call 2 is returned unchanged		`FALSE`

Table 9.6: Trace of `equal(T, L)`, *T = (a, (b, c)), L = (a, (b, d))*.

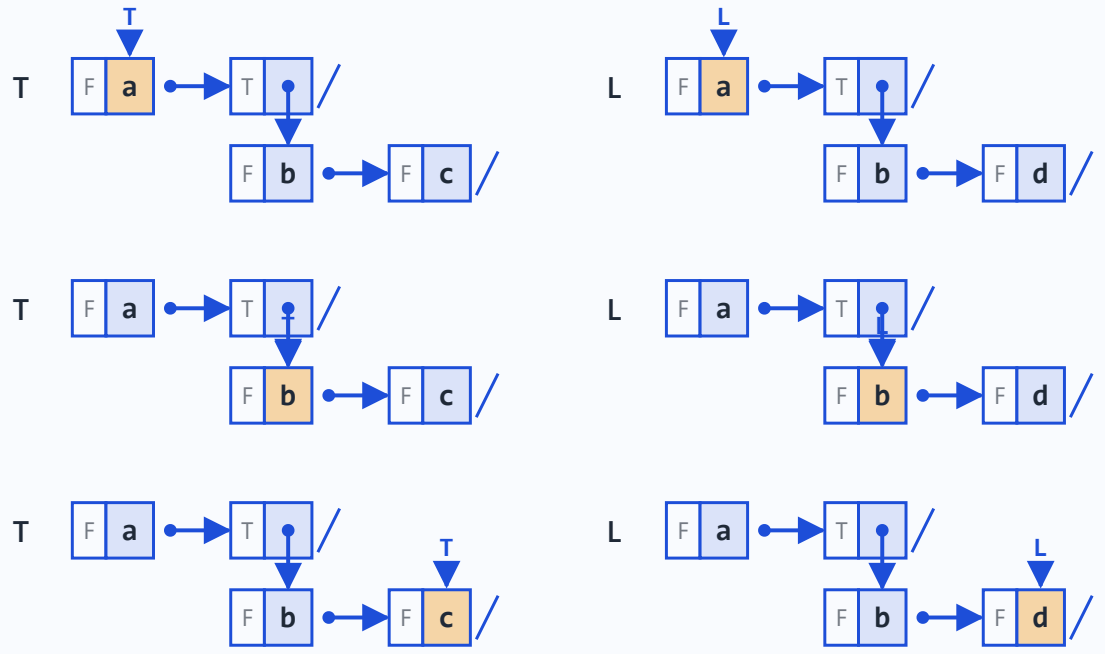


Figure 9.6: The pairs of elements compared during `equal(T, L)`. From top to bottom: the first atoms *a*/*a*; the first atoms of the sublists *b*/*b*; *c*/*d*, where the difference is found.

The function was called four times in total. This number was also measured with a copy instrumented with a counter. The effect of stopping at the first difference is seen here: call 2 does not look at the remaining elements of *T* and *L* at all, because the sublists did not come out equal. If the two lists are equal, all elements are walked and the number of calls is *1 + n + s*, as in `copy`. In the worst case the time is linear in the size of the shorter of the two lists: *O(n)*. The experiment also confirmed the following cases: *(a, (b, c))* equals a separately built twin of itself; it is not equal to *(a, (b, c), d)* (a difference in length); it is not equal to *(a, b, c)* (a difference of kind at the second element); two empty lists are equal.

9.5.6 A shorter version and shared lists

```
bool same(list_pointer T, list_pointer L) {
    if (T == L) return true;        // same node/both NULL
    if (!T || !L || T->tag != L->tag) return false;
    if (T->tag ? !same(T->dlink, L->dlink)
               : T->data != L->data)
        return false;
    return same(T->link, L->link);
}
```

This version uses C's `bool` type, and each condition is one line. The first line carries a new shortcut: if `T == L`, the two pointers point to the same element, or both are `NULL`. In both cases the lists are equal and there is no need to descend into them. The second line eliminates the first differences: if

one is empty or the tags differ, the result is false. The third line compares the contents of the elements. The conditional operator here selects not a value but a condition: in a T element, the sublists being unequal, and in an atom element, the values being different, gives a false result. The last line moves on to the rest.

The last line has one more property. The call `return same(T->link, L->link);` is the last thing the function does; no computation remains to be done after it returns. Such calls are called **tail calls**. With optimization on, compilers can turn a tail call into a jump, and moving along `link` then takes no space on the stack. The C standard does not guarantee this; but the way it is written allows it.

The real value of the shortcut shows in shared lists. On lists that are shared but not recursive, both routines give the right result. Sharing does not break equality; the same sublist is merely walked several times. `same` does not descend where it sees the same address on both sides. For example, when *B* is compared with itself, the first line returns true at once. When *B* is compared with a copy of itself that preserves sharing, the shared sublist in the copy is walked once.

The resistance of a program is found with inputs that try to break it. "I pressed the key and it worked" is not an adequate test. Knowing on which lists a routine is correct and on which it is wrong requires much broader knowledge than running it on a given example. Writing a proof lays this knowledge out explicitly. For `equal` this knowledge is as follows: correct on lists with no sharing and no recursion; correct on shared lists but doing needless work; not terminating on recursive lists.

9.5.7 Traversing and printing, and a faulty version

The third question asks for a routine that traverses the list and prints the atoms in order. A faulty version often seen in exams is the following:

```
void print_atoms( list_pointer L ){
    if(L)
        if(!L->tag)
            printf("%c", L->data);
        else
            print(L->dlink);
        print( L->link );
}
```

There are two errors in this code. The first is the name: the routine is named `print_atoms`, but the recursive calls call `print`. No such function is defined. Since C99, calling an undeclared function is an error. In the experiment the compiler stopped with the error "call to undeclared function 'print'".

The second error is more insidious and appears once the name is fixed. There are no braces after `if (L)`. An `if` without braces covers only the *single* statement that follows it. Here that statement is the whole inner `if-else`. The call `print( L->link );` on the last line is outside the `if (L)`; even though its indentation shows it inside, C does not look at indentation. This line also runs when `L ==`

`NULL`, and `NULL->link` is read. The compiler points to this with two warnings: "misleading indentation" and "add explicit braces to avoid dangling else". These warnings appear with `-Wall`; compiling without `-Wall` is therefore harming oneself.

When the name error was fixed and the code was run with *(a, (b), c)*, `ab` was printed to the screen and then the program crashed. After *b* was printed, `print_atoms(b->link)`, that is, `print_atoms(NULL)`, was called; the last line of this call tried to read `link` through a `NULL` pointer. UBSan caught this access with the message "member access within null pointer of type 'struct list_node'". The compiler also gave the warning "all paths through this function will call itself". According to this warning, every path of the routine calls itself at least once, that is, no call can end without recursion. This warning says directly that the base case is missing. The correct way to write it puts both calls inside `if (L) { … }`:

```
void print_atoms(list_pointer L) {
    if (L) {
        if (!L->tag)
            printf("%c", L->data); // print an atom
        else
            print_atoms(L->dlink);
        print_atoms(L->link);
    }
}
```

Common mistake

Trusting the indentation. In C a block is formed with braces; indentation is only for the reader. Putting braces even around single-statement `if` bodies prevents a line added later from being left outside by accident. Compiling with `-Wall -Wextra` shows most bugs of this kind before the program is even run.

9.5.8 Code trace: `print_atoms(L)`

Let *L = (a, (b), c)* be traversed. The second element is a sublist with a single element. Table 9.7 shows the calls and how the output grows.

Step	Stack (the one on the right is running)	Action	Output
1	print_atoms(a)	atom: `'a'` is printed; call for the next element	`a`
2	print_atoms(a) · print_atoms((b))	sublist: first dive into it (`L->dlink`)	`a`
3	… · print_atoms((b)) · print_atoms(b)	atom: `'b'` is printed; call for the next element	`ab`
4	… · print_atoms(b) · print_atoms(NULL)	`L == NULL`: end of the level, return at once	`ab`

Step	Stack (the one on the right is running)	Action	Output
5	print_atoms(a) · print_atoms((b)) · print_atoms(c)	returned from the sublist; moved to *c* through the link of `(b)`; `'c'` is printed	`abc`
6	… · print_atoms(c) · print_atoms(NULL)	end of the level; all the calls return in turn	`abc`

Table 9.7: Trace of `print_atoms(L)`, *L = (a, (b), c)*.

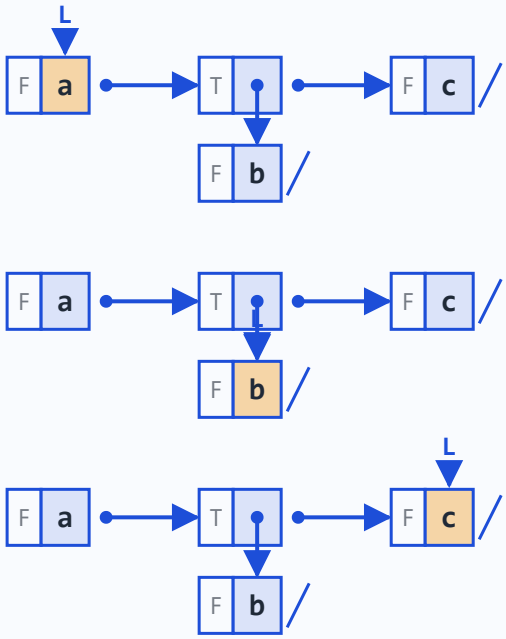


Figure 9.7: Three moments of `print_atoms` on *(a, (b), c)*: while *a* is being printed, while descending into the sublist and printing *b*, and after moving on to *c* at the same level. The label `L` shows the parameter of the call that is running at that moment.

At step 5 the move to *c* is made from within the call for the element `(b)`. `print_atoms(b)` is no longer on the stack; that call has finished after returning with `NULL`. But `print_atoms((b))` is still running, and its next job is to follow its own `link`. Atoms are printed from left to right regardless of depth; the output is `abc`. Every element is visited once; with the count from `copy`, the number of calls is $1 + n + s = 1 + 4 + 1 = 6$ and the time is $\Theta(n)$.

Something interesting happens when the same routine is applied to a list in tree form. Figure 9.8 shows the list that holds the tree *A(B(E(K, L), F), C(G), D(H(M), I, J))*. Every node of the tree is a sublist: the head of the sublist is the node's label and the remaining elements are its children. Children that are leaves are written directly as atoms, and children that have children are again written as sublists. In linear form the list is *(A, (B, (E, K, L), F), (C, G), (D, (H, M), I, J))*.

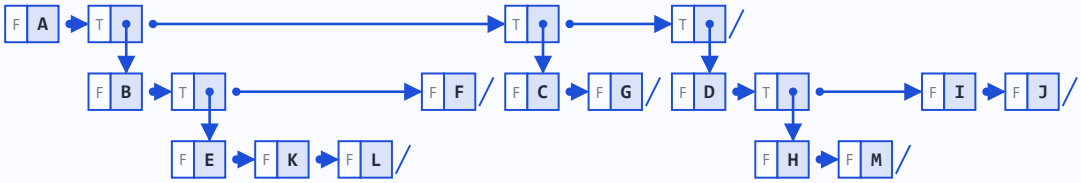


Figure 9.8: The generalized-list view of the tree *A(B(E(K, L), F), C(G), D(H(M), I, J))*. The first atom of every sublist is the label of a tree node, and the remaining elements are the children of that node. Elements on the same level are followed along one row.

`print_atoms` prints `ABEKLFCGDHMIJ` for this list. This order is no coincidence. Each sublist first prints its own head, that is, the label of the node, and then walks its children from left to right by the same rule. This is the **preorder** traversal of trees: first the root, then the subtrees in order. The list routine has thus run a tree algorithm without knowing anything about trees. This is exactly the gain of moving data to a well-known structure.

9.6 Depth

How deeply nested a generalized list is determines how much space the recursive routines that process it will occupy on the stack. In a list that holds a tree, the same measure corresponds to the height of the tree. In this section depth is first defined and then computed with two routines, one recursive and one iterative.

9.6.1 Recursive depth

Definition 9.3

The **depth** of the empty list is 0. The depth of a non-empty list L is 1 plus the largest of the depths of the sublists of L; if there are no sublists, this largest value is taken to be 0. Atoms contribute nothing to depth.

By the definition, the depth of *(a, b)* is 1: it has no sublists. The depth of *(a, (b, c))* is *1 + 1 = 2* and that of *(a, (b, (c)))* is 3. *(())* is an interesting example. This list is not empty; its only element is an empty sublist. Because the depth of the sublist is 0, the depth comes out as *1 + 0 = 1*. By the same reasoning the depth of *((()))* is 2. The question in the slides asks for this definition to be turned into code recursively:

```c
int depth(list_pointer L) {
    list_pointer p = L;  int m = 0;  // m: the max
    if (!L) return 0;
    while (p) {
        if (p->tag) {
            int n = depth(p->dlink);
            if (m < n) m = n;
        }
        p = p->link;
    }
    return m + 1;
}
```

The routine follows the definition almost word for word. For the empty list, 0 is returned. In a non-empty list the elements of the same level are walked with a `while` loop, and `m` holds the depth of the deepest sublist seen so far. Atom elements are skipped. At every sublist element the depth of the sublist is computed recursively and `m` is updated if necessary. When the loop ends, `m + 1` is returned: the deepest sublist plus this level.

Unlike copying, this routine does not recurse along `link`; it walks the same level with a loop. A recursive call is made only when descending into a sublist. The number of calls is therefore *1 + s*: one call from outside and one call for each sublist. Every element is visited exactly once inside a loop, and the time is $\Theta(n)$. The number of calls on the stack at the same time is at most the depth of the list. The extra memory of the routine is therefore $O(depth)$, and long but shallow lists cause no stack problem. In the experiment, the depth of a flat list of a million atoms was computed without trouble; the same list drove `copy` into a stack overflow.

The examples in the slides were confirmed by experiment: *(a, b)* → 1, *(a, (b, c))* → 2, *(a, (b, (c)))* → 3, *(())* → 1. In addition, *()* → 0, *((()))* → 2, and the exam-question list *(d, (c, (e, (m, k), ()), z, y, (x, (r), b, a,)))))* → 5 were found.

9.6.2 Code trace: `depth(L)`

For *L = (a, (b, (c)))* the routine descends three levels. The call at each level has its own `p` and `m` variables; the `m` at one level is independent of the `m` of the other levels. Table 9.8 shows the local variables of every call in separate columns.

Step	Running call	Action	`m` (level 1 / 2 / 3)
1	level 1	`p` = *a*; atom, skipped	0 / — / —
2	level 1	`p` = *(b, (c))*; sublist: level 2 is called	0 / — / —
3	level 2	`p` = *b*; atom. `p` = *(c)*; sublist: level 3 is called	0 / 0 / —
4	level 3	`p` = *c*; atom. `p = p->link` → `NULL`: the level is finished	0 / 0 / 0
5	level 3	`return m + 1` = 1	0 / 0 / —
6	level 2	`n = 1`; `m < n`: `m = 1`. `p` → `NULL`; `return 2`	0 / 1 / —

Step	Running call	Action	`m` (level 1 / 2 / 3)
7	level 1	`n = 2`; `m = 2`. `p` → `NULL`; `return 3`	2 / — / —

Table 9.8: Trace of `depth(L)`, *L = (a, (b, (c)))*. Every call has its own `m`.

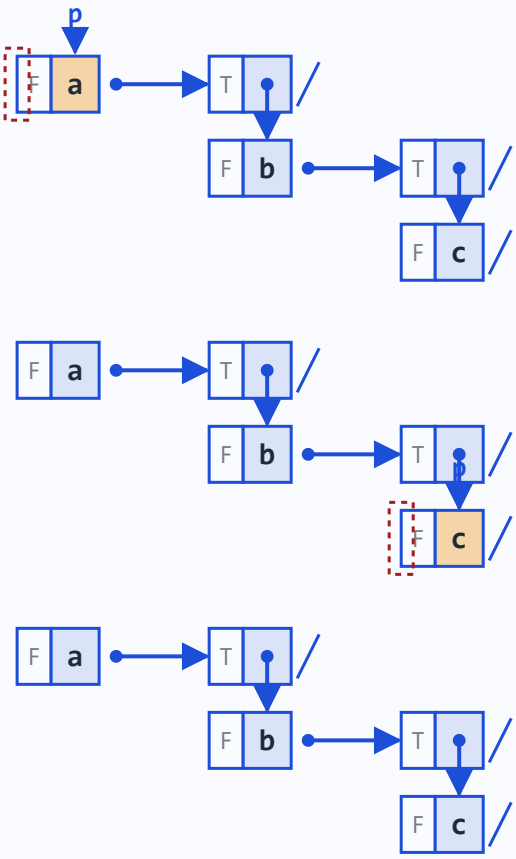


Figure 9.9: The element that `p` points to during `depth(L)`. At the top the level-1 call is at *a*. In the middle the level-3 call is at *c*. At the bottom all the levels have finished and the level-1 call returns 3.

What the trace shows is how recursion carries values. Information goes down only as an address: which sublist to look at. The answer comes back up as a return value. Each level knows only its own job: find the largest of the numbers coming from its sublists and add one. Nobody keeps the total depth anywhere, but the result forms correctly at the top.

9.6.3 A faulty iterative solution

An iterative answer given to the same question in an exam is the following:

```
int depth( tree_pointer L ){
    tree_pointer pop(); void push(tree_pointer);
    int max = 0;
    for( int current = 1, p = L ; p ; p->link ){
        if( p->tag ){ // enter the sublist
            push(p);
            p = p -> dlink;
            current++;
        }
        if( !p->link ){   // end of list, go up
            p = pop();
            if( max < current ) max = current;
            current--;
        }
    }
    return max;
}
```

The idea is this: `current` holds the current level and `max` the largest level seen. When descending into a sublist, the T element is pushed onto a stack and the level increases; when the end of a level is reached, an item is popped from the stack and the search goes back up. There is nothing wrong with the idea; a correct iterative solution follows the same path. What is wrong are the details, and all of them matter.

The code does not compile. The declaration `int current = 1, p = L` makes both variables `int`; in declarations separated by commas, the type is the one at the start of the line. When `p` is an integer, none of the expressions `p->tag`, `p->dlink`, and `p->link` is valid. The compiler gave seven errors; the first is the message "incompatible pointer to integer conversion initializing 'int'". The `p->link` in the third part of the `for` is not an assignment either. The value is computed and discarded; `p` never advances. Even if this error alone were fixed, the loop would stay at the same element forever. The right form is `p = p->link`.

Even if these are fixed, the code crashes on edge cases. When it descends into an empty sublist, `p = p->dlink` becomes `NULL` and right afterward `p->link` is read. When the list ends, `pop` is called even if the stack is empty. The search goes back to the T element popped from the stack itself, but not on to its sibling, that is, `T->link`. It therefore descends into the same sublist again. Moreover, the stack operations rely on a global stack outside.

This solution is criticized from two separate angles. First, the expected solution is recursive. Building a loop with the counters `current` and `max` is a first-year-level approach and moves away from the recursive definition of the structure. Second, global variables are forbidden. A routine written with a global stack or a global counter works while it is tested alone. But the same routine may become part of another component. For example, when the depths of two lists are computed nested, or when two threads call it at the same time, the shared variable is corrupted. The whole state of a routine must be in its own local variables.

9.6.4 Iterative depth with a local stack

An iterative solution can be written correctly. What is needed is to keep, explicitly in a local array, the call stack that recursion leaves to the compiler:

```
int iter_depth(list_pointer L) {
    list_pointer stack[100];   // local stack (depth<100)
    int top = 0, d = 1, m = 0;
    list_pointer p = L;
    if (!L) return 0;
    for (;;) {
        if (p == NULL) {             // level done
            if (top == 0) return m;
            p = stack[--top]->link;  // the node after T
            d--;
        } else {
            if (d > m) m = d;
            if (p->tag && p->dlink) {
                stack[top++] = p;    // descend
                p = p->dlink;
                d++;
            } else p = p->link;
        }
    }
}
```

The `stack` array holds the T elements of the sublists that have been entered; `top` is the number of items in the array. `d` is the current level and `m` the largest level seen. In each round of the loop, one of two cases holds.

- If `p` is not empty, we are on an element. This element is at level `d`, so `m` becomes at least `d`. If the element is a non-empty sublist, the T element is pushed onto the stack, `p` moves to the first element of the sublist, and `d` increases by one. If the element is an atom or an empty sublist, `p` moves to the next element on the same level.
- If `p` is empty, the end of a level has been reached. If the stack is also empty, the top level is finished and `m` is returned. Otherwise the T element entered last is popped from the stack and the search moves on to its *sibling*, `T->link`; `d` decreases by one.

All the defects of the faulty solution are fixed in this version. `p` is a pointer. An empty sublist is never entered, because `p->dlink` is checked separately. Popping is done only when the stack is not empty. After a pop, the search moves not to the popped element itself but to its sibling. There is no global variable. The contribution of an empty sublist to depth is also right. The only element of the list *(())* is seen at `d = 1` and `m` becomes 1; it does not become 2, because the sublist is not entered.

Round	`p`	`d`	`m`	`stack`	Action
1	*a*	1	0	empty	`m = 1`; atom: `p = p->link`

Round	`p`	`d`	`m`	`stack`	Action
2	T_1	1	1	empty	sublist: T_1 pushed; `p` = b, `d = 2`
3	b	2	1	T_1	`m = 2`; atom: next
4	T_2	2	2	T_1	sublist: T_2 pushed; `p` = c, `d = 3`
5	c	3	2	T_1, T_2	`m = 3`; atom: `p = NULL`
6	`NULL`	3	3	T_1, T_2	level finished: T_2 popped, `p = T2->link` = `NULL`, `d = 2`
7	`NULL`	2	3	T_1	T_1 popped, `p = T1->link` = `NULL`, `d = 1`
8	`NULL`	1	3	empty	stack empty: `return 3`

Table 9.9: Trace of `iter_depth(L)`, $L = (a, (b, (c)))$. Each row is the state at the start of one round of the loop; $T_1 = (b, (c))$, T_2 = the element (c).

The values in the table are the same as the output of a copy of the routine with a print line added to every round of the loop. The correctness of the routine was also tested in two ways. On twelve example lists and on 20,000 random lists, `iter_depth` and `depth` gave the same result. The time is again $\Theta(n)$: every element is visited once, and every T element is pushed once and popped once.

The fixed size of the stack array is the weak point of this solution. 100 cells are enough to descend into at most 100 nested sublists; that is, lists with depth up to 101 are computed correctly. In a deeper list, `stack[top++] = p` writes outside the array. In the experiment, on a chain of depth 102, UBSan stopped the program with the error "index 100 out of bounds for type 'list_pointer[100]'". A robust version either returns an error when `top == 100` or grows the stack with `malloc` and `realloc`. The recursive version has a limit too, but that limit is the size of the call stack and is much higher: in the same experiment, `depth` ran without trouble on a chain of depth 5000.

Common mistake

Returning to the T element popped from the stack in iterative traversal. The sublist of the T element is already finished. The traversal must continue from T's sibling, that is, from `T->link`. Code that returns to T itself descends into the same sublist again and can never leave the loop.

9.6.5 Thinking recursively and extreme examples

When the two solutions of the depth question are put side by side, the difference is clear. The recursive version is ten lines and follows the definition exactly. The iterative version has to do by hand what recursion does by itself: keep track of which sublist we are in and where to return from there. This version is longer, carries a fixed limit, and is much easier to get wrong when writing.

Thinking recursively is a habit and takes time to acquire. A one-semester course is not enough for this foundation. For every recursive call one should ask, "If I give it this list, what will it return to me?", and not move on to the rest of the routine without writing down the answer. Running the code step

by step with a debugger and following which value goes where at every return is the best way to see the answer to this question. The trace tables in these notes do the same job on paper.

Code that works only on the given example is not enough. A routine may give the right result on the given example and crash on others. Correctly working code is code that works correctly on extreme examples too. For generalized lists the extreme examples are well defined. Table 9.10 collects these examples and the behavior seen in the experiments.

List	What it tests	Observed
()	empty list; `NULL` check	`depth` 0; `copy` `NULL` ; `print_linear` `()`
(a)	single element	depth 1; inversion gives the same list
(())	empty sublist; a T element with `dlink == NULL`	depth 1; `print_linear` `(())`
((()))	nested empty sublists	depth 2
long single-level list	recursion along `link`	all routines worked on $5 \cdot 10^4$ atoms; `copy` and `equal` stopped with a stack overflow on 10^5 atoms and `print_atoms` on 10^6 atoms; `depth` and `invert` worked on 10^6 atoms
a single deep chain *(((... (a)...)))*	recursion along `dlink` ; fixed stack array	`depth` worked at depth 5000; `iter_depth` overflowed the array beyond 101

Table 9.10: Extreme example lists and the behavior of the routines (experiment: `-O0` , Windows, default 1 MB stack).

The fifth row of the table shows an important detail. `copy` , `equal` , and `print_atoms` also recurse along `link` ; so on a long level every element holds one call on the stack. `depth` and `invert` , on the other hand, walk the same level with a loop and take stack space only in proportion to the nesting. Which routine can safely be used on which lists is also summarized by the extra-memory column in the complexity table. That `print_atoms` endures longer lists than `copy` is because each of its calls takes less space on the stack.

A data structures course is the backbone of computer science. It is not a course that can be summed up as "isn't it just lists and trees?"; summing up brain surgery as "these blood vessels" would be a similar injustice. Knowing lists and trees does not mean having learned data structures. The measure of learning is being able to write the routines that build, copy, compare, and traverse the structure. A solid mathematical background is also needed for this: discrete mathematics, probability and statistics, random processes. An education that starts early with programming assignments teaches fast coding, but carries the danger of neglecting theory. The courses of the department cannot give everything; topics found lacking can be noticed early and made up with courses taken from the mathematics or statistics departments. R. Johnsonbaugh's *Discrete Mathematics* is among the books recommended for this background.

9.7 Inversion

Inverting a generalized list means reversing the order of the elements on every level. Inversion is applied to every level: the inverse of the list *L = (a, (b, c))* is *((c, b), a)*. If only the top level were reversed, the result would be *((b, c), a)*. This operation corresponds to taking the mirror image of the picture in a list that holds a tree or a nested shape.

9.7.1 Three pointers and two extra lines

The classic routine that reverses a singly linked list in place uses three pointers. `p` holds the head of the part not yet reversed, `q` the head of the reversed part, and `r` the previous `q`. In each round the element at `p` is taken onto the head of the reversed part. For the generalized list, only two lines are added to this routine:

```
void invert(list_pointer *L) {
    list_pointer r, p = *L, q = NULL;
    while (p) {
        if (p->tag)                 // not an atom
            invert(&p->dlink);
        r = q;
        q = p;
        p = p->link;
        q->link = r;
    }
    *L = q;
}
```

The four assignments of the loop are the same as in the classic routine. `r = q` saves the element reversed in the previous round. `q = p` moves to the element to be reversed in this round. `p = p->link` advances to the rest of the list; this assignment must be made before the link is changed, otherwise the rest of the list is lost. `q->link = r` turns the link of the element backward. When the loop ends, `q` points to the last element of the old list, that is, the first element of the new list, and `*L = q` makes it the head of the list.

The two added lines first reverse the sublist itself when a sublist element is reached. Here it is the address of `p->dlink`, not its value, that is given to `invert`. The reason is that inversion changes the head of the list. When *(b, c)* is reversed, the first element of the sublist becomes *c*, not *b*. The `dlink` field of the T element must point to this new head. The inner call writes `*L = q` on its last line. Because `L` is the address of the `dlink` field of the T element, this assignment updates that field directly. The outer call has nothing left to do after the return. Had the value been given, the inner call would have updated its own local copy. The T element would have kept pointing to the old head, that is, to *b*, now the last element, and *c* would have become unreachable and leaked.

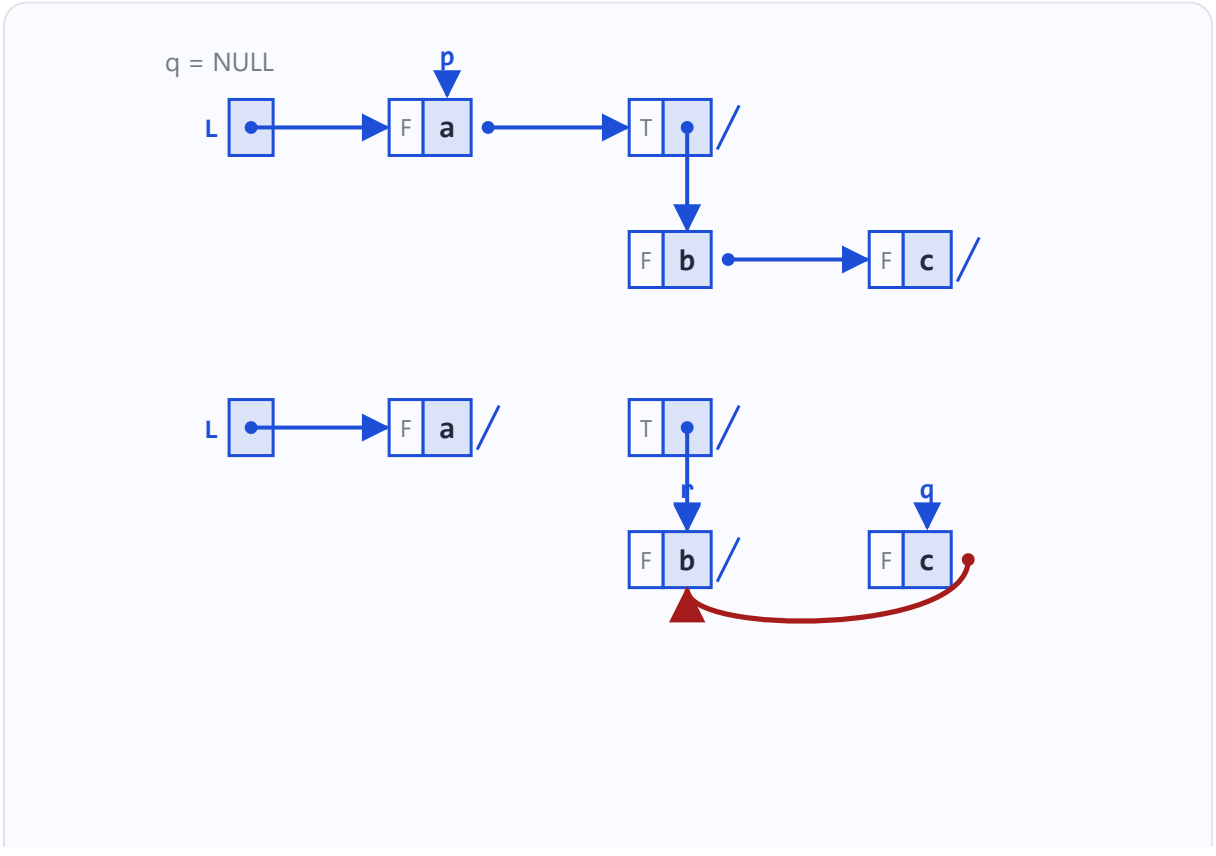
For the same reason the call from outside is also made by address: `invert(&L)`. The head of the top level changes too, and the caller's variable `L` must be updated. Passing the list by value to a routine that changes the head is one of the most frequent mistakes with linked lists.

9.7.2 Code trace: `invert(&L)`

Let *L = (a, (b, c))* be inverted. Table 9.11 gives the twelve steps of the slides. T is the second top-level element, that is, the sublist element.

Step	Call	Action	`r`	`q`	`p`
1	outer	`p = *L`, `q = NULL`	?	NULL	*a*
2	outer	*a* is an atom; four assignments: the link of *a* became `NULL`	NULL	*a*	T
3	outer	T is a sublist: `invert(&p->dlink)`	NULL	*a*	T
4	inner	`p = *L` = *b*, `q = NULL`	?	NULL	*b*
5	inner	the link of *b* became `NULL`	NULL	*b*	*c*
6	inner	the link of *c* was turned to *b*	*b*	*c*	NULL
7–8	inner	`p == NULL`: the level is finished; `*L = q`: the `dlink` of T is now *c*	*b*	*c*	NULL
9	outer	returned from the inner call; T already points to the new head	NULL	*a*	T
10	outer	the link of T was turned to *a*	*a*	T	NULL
11–12	outer	`p == NULL`; `*L = q`: the head of the list is T	*a*	T	NULL

Table 9.11: Trace of `invert(&L)`, *(a, (b, c)) → ((c, b), a)*.



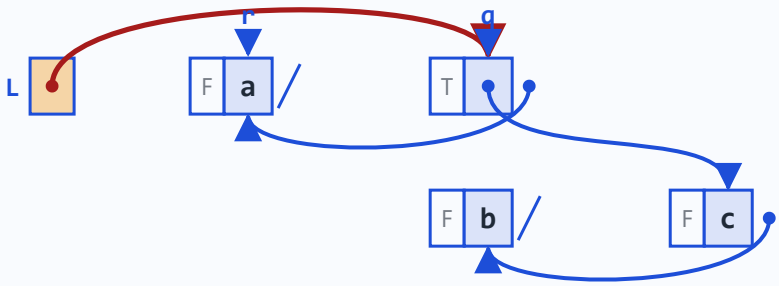


Figure 9.10: Three moments of `invert(&L)`. Top, the start: $L \to a \to$ T, with $b \to c$ below T. Middle: at the top level the link of a has been cut, and in the sublist the link of c has been turned to b. Bottom, the result: $L \to$ T $\to a$; the sublist of T is $c \to b$.

The routine allocates no element and frees no element; it only changes links. In the experiment the `malloc` counter never increased during inversion. The time is $\Theta(n)$: every element becomes `p` once. The number of calls is $1 + s$, and because the same level is walked with a loop, the extra memory is $O(depth)$. The experiment gave these results: *((a, b), ((c, d), e)) → ((e, (d, c)), (b, a))*; *(a, (()), b) → (b, (()), a)*; *(a, (b, (c))) → (((c), b), a)*; *()* and *(a)* are unchanged. Inverting any list twice gave back the first list; this property was also tested on 20,000 random lists.

This solution is the generalization of a well-known routine with a small addition. The same result could also be reached by brute force. For example, collecting the elements of each level into an array and relinking them in reverse order, or copying the elements one by one onto the head of a new list, would do the job. These ways also invert, but they spend needless memory and work; like a storm, they tear down everything in their path and rebuild it. The three-pointer routine works in place and touches each link in the structure only once.

9.8 Building a tree with a list

A tree consists of a root node and zero or more subtrees, each of which is again a tree. This definition is also recursive and very similar to the definition of the generalized list. This similarity makes it possible to store a tree with a one-line text and with a list built from that text.

9.8.1 The linear form of a tree

In the linear form of a tree every node is written with its label; if it has children, they are listed right after the label, in parentheses, separated by commas. *C(G)* says that C is the parent of G. *A(B..., C..., D...)* says that A is the parent of B, C, and D. Figure 9.11 gives the tree in the slides and its linear form.

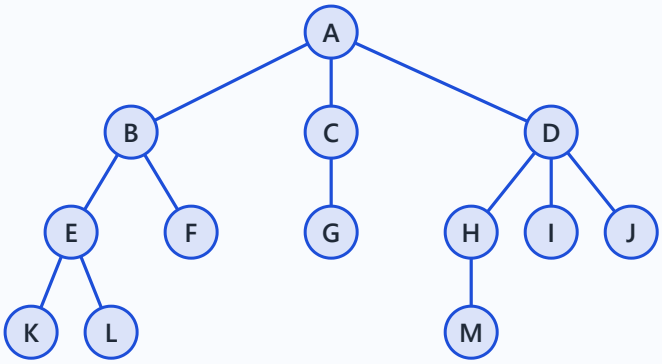


Figure 9.11: The tree whose linear form is *A(B(E(K, L), F), C(G), D(H(M), I, J))*. A has three children; K, L, F, G, M, I, and J are leaves.

The text carries the whole structure of the tree. Which node is the child of whom and the order of siblings can be read unambiguously from the text. For this much data, a one-line text suffices. The notation is also a choice: a period can be used instead of a comma, and braces instead of ordinary parentheses. What matters is that the routines that go from text to structure and from structure to text can be written. When both directions are provided, nobody outside needs to know the structure inside the program.

For the transition from text to list, the tree is held as a list whose first element is the root label: the tree *A(B, C)* becomes the list *(A, B, C)*. A child that has children is again a sublist of this form. The tree in the slides thus turns into the list *(A, (B, (E, K, L), F), (C, G), (D, (H, M), I, J))*. This transformation amounts to replacing every piece *X(* in the tree text with *(X,*, and the linear-form reader builds the list directly.

9.8.2 The right–left problem in binary trees

In a general tree the order of the children is enough information: the first child, the second child, and so on. In a **binary tree**, however, each node has at most two children, and they are distinguished as *left* and *right*. In a node with a single child, whether this child is left or right is part of the structure. If two trees differ only in this respect, they are different trees.

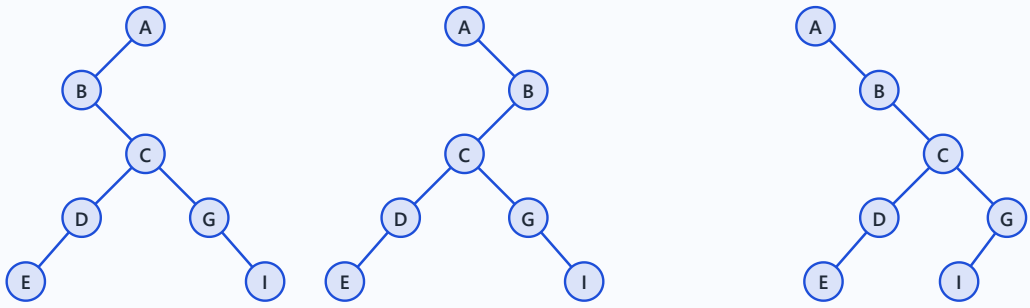


Figure 9.12: Three different binary trees with the same linear form, *A(B(C(D(E), G(I))))*. In the left one, B is the left child of A and C is the right child of B. In the middle one, B is the right child and C the left. In the right one,

B and C are right children; I is also the left child of G.

Figure 9.12 shows the problem. The linear form of all three trees is *A(B(C(D(E), G(I))))*. The text says that B is the only child of A but cannot say on which side. The same ambiguity exists for C, E, and I. The linear form loses information in a binary tree.

The solution is to add the missing information to the text. Either an L or R mark is kept next to each node, or a special symbol is written in place of an empty child. In the second way, the left tree can be written as *A(B(, C(...)),)*: the left child of B is empty, so nothing is written before the comma. The abbreviation *A(B(, C(...)))* in the slides shows this idea. The list of children of every node then has exactly two positions and no information is lost.

9.8.3 From tree to text: `print_tree`

Two of the assignments in the slides are related to this topic. The first asks for the memory layout and the tree structure of the list *A(B(E(H, I(J, K)), F), C(G), D)* to be drawn. The second asks for a routine that outputs the list notation when the tree structure is given.

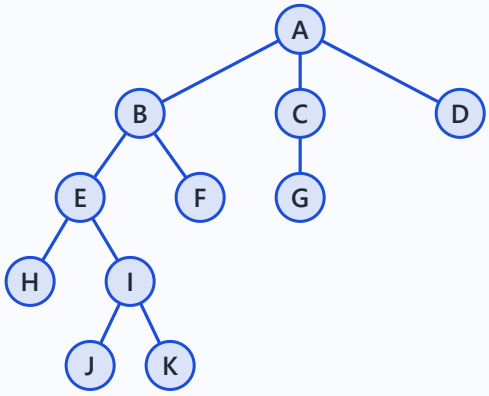


Figure 9.13: The tree of the first assignment: *A(B(E(H, I(J, K)), F), C(G), D)*. I has two children (J, K); D is a leaf.

The tree structure is given in Figure 9.13. Its list is *(A, (B, (E, H, (I, J, K)), F), (C, G), D)*. In the memory layout the top row consists of the elements A, T, T, D. Below the first T are B, T, F; below that T are E, H, T; at the bottom are I, J, K. Below the second T are C and G. This list has *4 + 3 + 3 + 3 + 2 = 15* elements and depth 4. The solution of the second assignment is this:

```
void print_tree(list_pointer L) { // L: (root, subtrees...)
    printf("%c", L->data);          // root label
    if (L->link) {
        printf("(");
        for (list_pointer p = L->link; p; p = p->link) {
            if (p->tag) print_tree(p->dlink);
            else printf("%c", p->data);   // leaf
            if (p->link) printf(",");
        }
        printf(")");
    }
}
```

The parameter of the routine is a list whose first element is the root label. First the root label is printed. If there are elements in the list after the root, that is, if the node has children, an opening parenthesis is printed and the children are walked in order. A child that is a sublist is a subtree and is printed with the same routine. A child that is an atom is a leaf and only its label is printed. A comma is put after every child except the last. At the end the parenthesis is closed.

Unlike `print_atoms`, this routine prints parentheses and commas as well. But it opens the parenthesis not at the head of each sublist but *after* the root label. This small difference converts list notation to tree notation. In the experiment, the list of the tree in the slides gave the output `A(B(E(K,L),F),C(G),D(H(M),I,J))` and the list of the tree in the assignment gave `A(B(E(H,I(J,K)),F),C(G),D)`. Both are the same as the linear forms of the trees. For the one-node list *(A)*, only `A` is printed. The routine is not defined for the empty list; for an empty tree `L->data` would be read and the program would crash. The caller must check that the tree is not empty.

The time is $\Theta(n)$: every element is printed once. Because the same level is walked with a loop, the stack depth equals the height of the tree.

9.9 Sharing, reference counting, and garbage collection

Shared lists save memory; the same sublist is used in many places without being copied. But this saving has a price. When can a shared sublist be freed? If it is freed because one of the places pointing to it no longer uses it, the other places point to returned memory. If it is never freed, memory leaks.

9.9.1 The header node and the reference count

In the solution proposed by Horowitz and Sahni, every list begins with a **header node**. The header node is not an element of the list; it is the identity of the list. Everyone who points to the list points to the header node, not to the first element. A field `ref` is kept in the header. This field is the number of pointers that point to the header; it is called the **reference count** of the list. In the example, the field `int ref;` has been added to the element structure.

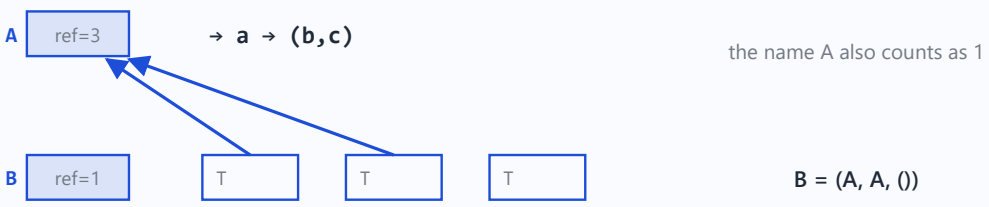
```
// header ref: how many pointers point to this list
void list_free(list_pointer *L) {
    list_pointer p, q;
    if (*L == NULL) return;
    if (--(*L)->ref > 0) {      // still in use
        *L = NULL; return;
    }
    p = (*L)->link;             // first real node
    free(*L);
    while (p) {
        if (p->tag) list_free(&p->dlink);
        q = p; p = p->link; free(q);
    }
    *L = NULL;
}
```

`list_free` announces that a pointer to the list will no longer be used. First it decreases the counter by one. If the counter is still greater than zero, others are using the list as well. The routine only sets the caller's pointer to `NULL` and returns. If the counter has dropped to zero, nobody points to the list any more. The header is freed, and then the elements are freed in turn. For sublist elements, the sublist itself is freed with the same routine. This is a chain rule: if the sublist has other pointers to it, its counter merely decreases and it stays in memory.

Three details of the routine matter. First, the address of the first element is taken into `p` before the header is freed; after `free(*L)`, `(*L)->link` cannot be read. Second, in the loop the link of each element is read before `free`. Third, the parameter is `list_pointer *L` and the routine writes `*L = NULL` in every case. The caller's pointer thus does not remain as a dangling pointer. Keeping the counter right is up to the programmer: every piece of code that creates a new pointer to the list must increase the counter by one, and every piece of code that drops a pointer must call `list_free`.

9.9.2 Step by step: `B = (A, A, ())`

Let *A = (a, (b, c))* and *B = (A, A, ())*. Three pointers point to the header of *A*: the variable named `A` and the first two T elements of *B*. Its counter is therefore 3. Only the variable `B` points to the header of *B*, and its counter is 1. The empty list that the third element of *B* points to also has its own header, and its counter is 1.



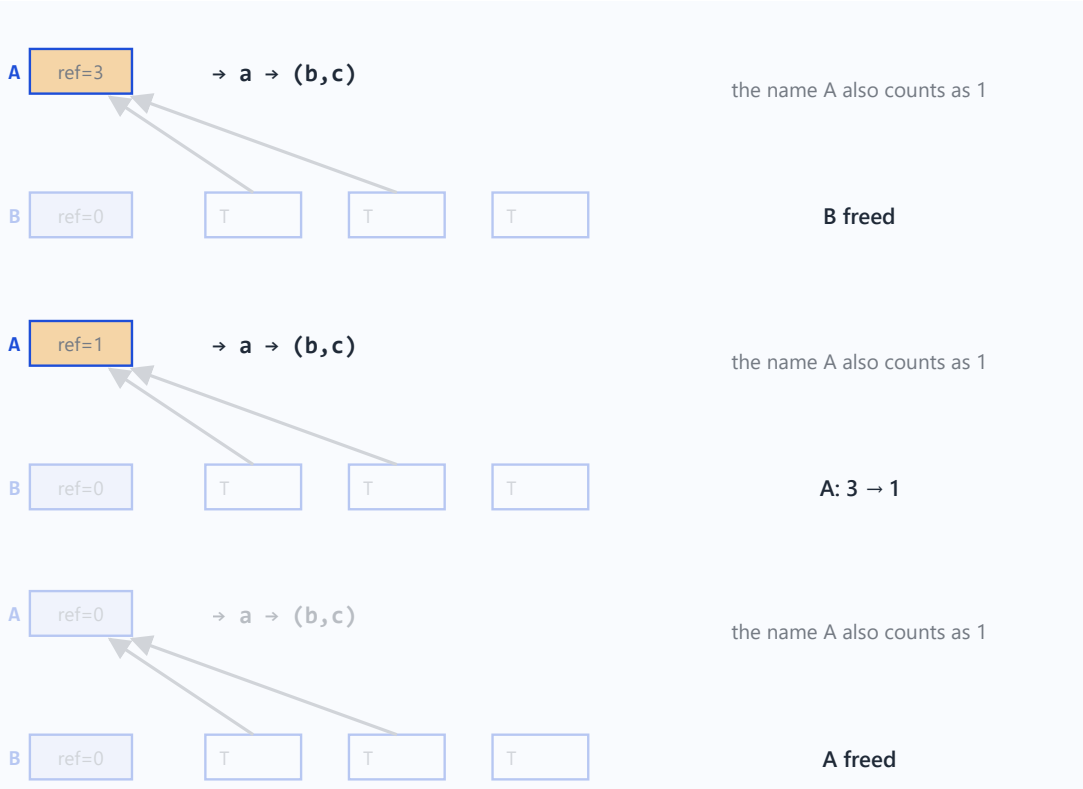


Figure 9.14: Reference counting step by step. From top to bottom: at the start the counter of *A* is 3 and that of *B* is 1; `list_free(&B)` lowers the counter of *B* to 0; the two T elements of *B* lower the counter of *A* from 3 to 1, and *A* is not freed; `list_free(&A)` lowers the counter to 0 and *A* is freed.

1. `list_free(&B)`: the counter of *B* goes from 1 to 0. The header is freed and the walk over the elements begins.
2. The first T element calls `list_free(&p->dlink)`. The counter of *A* goes from 3 to 2; because it is not zero, *A* is not touched, and only the `dlink` of the T element becomes `NULL`. The T element is freed.
3. The second T element does the same: the counter of *A* goes from 2 to 1.
4. The third T element makes a call for the header of the empty list. The counter of that header goes from 1 to 0; the header is freed, and since it has no first element, the loop never runs. The T element is freed. `B` is now `NULL`.
5. When the program frees *A* as well, `list_free(&A)` lowers the counter from 1 to 0. The header of *A*, *a*, the T element, and the header of *(b, c)*, *b*, and *c* are freed.

This scenario was run with counting `malloc` / `free` wrappers. The structure uses 11 blocks: 6 for *A* (two headers and four elements), 1 header for the empty list, and 1 header and 3 elements for *B*. `list_free(&B)` freed 5 blocks and left the counter of *A* at 1. `list_free(&A)` freed the remaining 6 blocks. A total of 11 allocations and 11 releases were counted; no block was freed twice and there was no leak. Sharing was managed safely.

The same idea can be applied to copying as well. Instead of copying a shared sublist, its counter is increased by one and the new list points to the same header. Copying thus becomes both faster and cheaper in memory. The counterpart is that care must be taken when lists are modified: a change made in a shared list is visible everywhere that points to it.

9.9.3 The limit of reference counting and garbage collection

Reference counting falls short with recursive lists. Two pointers point to the header of *C = (a, C)*: the variable `C` and the T element of the list itself. The counter is 2. When the program calls `list_free(&C)`, the counter drops to 1 and the list is not freed. But now no variable of the program can reach this list. The only pointer that could bring the counter down to zero is inside the list itself. The list stays in memory forever. This situation was observed in the experiment: after `list_free(&C)`, `C` became `NULL`, the counter stayed at 1, and none of the 3 allocated blocks was freed. The same problem arises with every cycle, such as two lists pointing to each other.

Another method is needed to collect cycles as well: **garbage collection**. Its best-known form is the **mark and sweep** method, which consists of two phases:

1. *Marking.* The search starts from the program's variables, that is, from the **root pointers**. All the elements reachable from them are traversed and the mark bit of each is set. The traversal stops when it sees an element a second time; cycles are therefore no problem.
2. *Sweeping.* The memory in which the elements live is scanned from beginning to end. Every unmarked element is garbage that cannot be reached from anywhere and is added to the list of free nodes. The marks of marked elements are cleared for the next collection.

In the example *C = (a, C)*, after the variable `C` is deleted, none of the root pointers reaches the list. The marking phase never touches the elements of the list and the sweep collects them. Unlike with reference counting, the cycle is no obstacle here.

Garbage collection has a cost too. While the collector runs, the program stops or slows down; in a large memory this pause is noticeable. Every element needs space for a mark bit. LISP, Java, and Python use refined variants of this idea. Python, for example, first counts references and finds cycles with a separate collector. C has no automatic garbage collection. The responsibility for copying, sharing, and freeing lies entirely with the programmer. A C programmer therefore must know how each routine they write behaves on shared and recursive lists.

9.10 Exam questions

The last section of the slides gives three exam questions asked on generalized lists. All three are applications of the pattern described in these notes.

9.10.1 Element count and sublist counters

Question. The element structure has been given an `int count` field. Write recursively the routine that writes into the counter of every element whose tag is `TRUE` the number of elements beneath it, and returns the total number of elements.

```
int count_nodes(list_pointer L) {
    int total = 0;             // an empty list has 0 nodes
    for (; L; L = L->link) {
        if (L->tag) {           // is there a sublist?
            L->count = count_nodes(L->dlink);
            total += L->count;  // count the sublist's nodes too
        }
        total++;                // the current node
    }
    return total;
}
```

The routine walks the same level with a `for` loop and accumulates the elements in `total`. In an empty list the loop never runs and 0 is returned. Every element is counted once for itself. At a sublist element, the number of elements in the sublist is additionally found recursively, written into the `count` field of the element, and added to the total. The counter of a T element thus counts only the elements beneath it; the T element itself is counted one level up.

For *(a, (b, c))* the counter of the T element is 2 and the return value is 4: *a*, the T element, *b*, and *c*. The experiment confirmed this and more complex lists. For *((a, b), ((c, d), e))* the result is 8 and the counters of the T elements are 2, 4, and 2. For *(())* the result is 1 and the counter of the single T element is 0. The list in the last exam question of the slides has 19 elements.

9.10.2 Printing in linear form

Question. Write the recursive routine that prints a generalized list in memory in linear form.

```
void print_linear(list_pointer L) {
    printf("(");
    for (; L; L = L->link) {
        if (L->tag) print_linear(L->dlink);
        else printf("%c", L->data);
        if (L->link) printf(",");
    }
    printf(")");
}
```

Each call of the routine prints one list, so the opening and closing parentheses are there in every call. In between, the elements are walked: a sublist element is printed with the same routine, and an atom with just its value. A comma is put after every element except the last. For an empty sublist the call is made with `NULL`; the loop never runs and `()` is printed. Empty sublists therefore do not disappear: *(a, (()), b)* is reproduced exactly.

This routine and the linear-form reader are inverses of each other. When a text that was read is printed, the same text must come out. This property was tested on 20,000 random texts and no difference was found. A pair of routines that convert a structure to text and rebuild it from text is the basis for saving the structure, sending it over a network, and testing it.

9.10.3 Drawing a list

Question. Draw the generalized-list view that corresponds to the linear notation *L = (d, (c, (e, (m, k), (()), z, y, (x, (r), b, a, ()))))*. Elements that follow one another on the same level must be followable along one row.

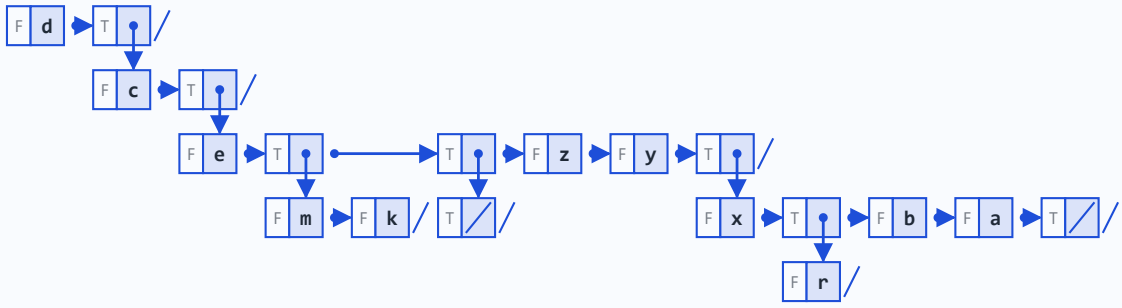


Figure 9.15: The solution of the exam question. Each row is a level; the arrows going down descend into sublists. Below the element *(())* there is a single T element whose sublist link is null. The last element of the bottom level, *()*, is a T element whose sublist link is null.

The drawing is made by counting parentheses. The outermost list has two elements: *d* and a sublist. That sublist also has two elements: *c* and a sublist. The third level is the most crowded: six elements, *e*, *(m, k)*, *(())*, *z*, *y*, and a last sublist. *(m, k)* is a sublist with two atoms. *(())* is a one-element list, and that single element is the empty list. In the drawing, below a T element comes a T element whose sublist link is null. On the fourth level there are *x*, *(r)*, *b*, *a*, and *()*. *()* is the last element and is a T element; its sublist link is null.

The most common mistake in this question is confusing *()* with *(())*. *()* is an element, and a T box whose sublist link is `NULL` is drawn. *(())* is also an element, but its sublist link is not null: it points to a list of one element, and the only element of that list is *()*. The correctness of the drawing can be checked by counting the elements: the list has 11 atoms and 8 T elements, 19 elements in total; the depth is 5.

9.11 Complexity summary and measurements

Table 9.12 collects the costs of the routines examined in these notes. n is the total number of elements in the list; the elements of sublists are counted too.

Routine	Time	Extra memory	Description
`copy`	$\Theta(n)$	$O(depth + length)$*	no sharing, no recursion; plus $\Theta(n)$ for the copy itself
`equal` / `same`	$O(n)$	$O(depth + length)$*	stops at the first difference
`print_atoms`	$\Theta(n)$	$O(depth + length)$*	preorder
`depth` (recursive)	$\Theta(n)$	$O(depth)$	level by loop, sublist by recursion
`iter_depth` (iterative)	$\Theta(n)$	$O(depth)$	local stack; depth at most 101
`invert`	$\Theta(n)$	$O(depth)$	in place, no `malloc`
`array_insert` / `array_delete`	$O(n)$	$O(1)$	shifting; $O(1)$ at the end
`ll_insert` (position i)	$O(i)$	$O(1)$	no shifting

Table 9.12: Time and extra-memory cost of the routines.

The starred rows mark a subtlety. In routines that also make a recursive call along `link`, the stack depth depends not only on nesting but also on the number of elements on the same level. In a small list such as *(a, (b, c))* this difference is invisible, but in a single-level list of a million elements the stack descends to a depth of a million calls. Using a loop for `link`, that is, recursing only into sublists as in `depth` and `invert`, brings the extra memory down to $O(depth)$.

The time column was also tested with a measurement. Lists made of k sublists of k atoms each were built ($n = k^2 + k$) and the routines were run three times each in a program compiled with `-O2`. Table 9.13 gives typical values. The values vary from machine to machine; what matters is the way they grow.

n	`copy`	`depth`	`invert`
≈ 250 thousand	12–17	≈ 2	≈ 2
≈ 1 million	45–57	5–8	7–9
≈ 4 million	180–220	28–29	31–34

Table 9.13: Measured times (ms, approximate values of three runs; they vary from machine to machine).

As n quadruples, the times also roughly quadruple; this is linear growth. `copy` is six to seven times slower than the others, because it calls `malloc` for every element. Allocating memory is a far more expensive operation than following a link. There can be this much difference between two routines in the same $\Theta(n)$ class.

9.12 Common mistakes

1. **Assuming a shared list is unshared.** `copy` copies a shared sublist separately wherever it is pointed to; `free_list` returns it twice. The second is undefined behavior.
2. **Calling copy or equality on a recursive list.** For *C = (a, C)* the calls never end and the stack overflows.
3. **Forgetting braces in `if` statements.** In the faulty `print_atoms`, the last call also runs for `NULL` and the program crashes while reading `NULL->link`.
4. **Passing the list by value to a routine that changes the head.** Inversion needs `list_pointer *L`; for sublists, `&p->dlink` is passed.
5. **Confusing the empty sublist *()* with the empty list (`NULL`).** *()* is an element: the T element exists, only its `dlink` is `NULL`.
6. **Counting depth with a global variable.** When the routine becomes part of another component or is called in a nested way, the shared variable is corrupted.
7. **Reading a field of the union without checking the tag.** The `dlink` of an atom element is not an address.
8. **Returning to the T element popped from the stack in iterative traversal.** The place to continue is T's sibling, `T->link`.
9. **Writing the result of `realloc` directly into the same pointer.** If memory runs out, the old block is lost.

9.13 Exercises and solutions

The exercises and hints in the slides have been turned into full solutions below. All the solutions were compiled and tried on the example lists.

9.13.1 Atom count

Question. Write the routine that finds the number of atoms in a generalized list. The path suggested by the slides: 1 for an atom, the recursive result for a sublist; add along `link`.

```
int count_atoms(list_pointer L) {
    int k = 0;
    for (; L; L = L->link)
        k += L->tag ? count_atoms(L->dlink) : 1;
    return k;
}
```

The routine is a simpler sibling of `count_nodes`. The sublist element itself is not counted, only the atoms inside it. The list in the exam question has 11 atoms, and the tree list has 13 atoms (the 13 nodes of the tree). The time is $\Theta(n)$ and the extra memory is $O(depth)$.

9.13.2 Building a list from its linear form

Question. Write the routine that reads a linear-form text such as `"(a,(b,c))"` and builds the list. The path suggested by the slides: on seeing `(`, build a sublist; advance with `,`; return at `)`.

The solution to this question is the `parse` routine, described in detail in the section that introduced generalized lists. The routine carries the position in the text with `const char **`, calls itself for every sublist, and links the elements at the end with the `tail` pointer. A routine that builds the list is a very suitable topic for short questions in exams. Constantly thinking about how to write the routines that build, copy, and print the list makes this topic stick.

9.13.3 Copying that preserves sharing

Question. How is `copy` changed so that it preserves sharing? The path suggested by the slides: keep the sublists already copied (old address → new address) in a table.

```
#define TABLE_SIZE 100
typedef struct { list_pointer from, to; } pair;

// t[0..*n-1]: sublists already copied (old head -> new head)
list_pointer copy_shared(list_pointer L, pair t[], int *n) {
    list_pointer head = NULL, *tail = &head, q;
    int k;
    for (; L; L = L->link) {
        q = get_node();
        q->tag = L->tag;
        if (!L->tag) q->data = L->data;
        else {
            for (k = 0; k < *n && t[k].from != L->dlink; k++)
                ;                       // search the table
            if (k < *n) q->dlink = t[k].to;   // shared
            else {
                q->dlink = copy_shared(L->dlink, t, n);
                if (*n < TABLE_SIZE) {
                    t[*n].from = L->dlink;
                    t[*n].to = q->dlink;
                    (*n)++;
                }
            }
        }
        *tail = q;
        tail = &q->link;
    }
    *tail = NULL;
    return head;
}
```

The routine walks the same level with a loop and appends the elements at the end with the `tail` pointer. At a sublist element, the table is consulted first. If the old head of the sublist is in the table, the new element points to the copy in the table and the sublist is not copied again. If not, the sublist is copied and the pair is added to the table. For *B = (A, A, ())* the routine allocated 7 elements (`copy` had allocated 11). The first two elements of the copy point to the same list, and `same` finds the copy equal to the original list. The table has 3 pairs: *A*, *(b, c)*, and the empty list.

The limits of the solution are clear. Because the table is searched linearly, *O(t)* work is done for every sublist; with many sublists, a hash table is more suitable. If the table fills up, sharing is silently lost. On recursive lists the routine still does not terminate, because the record of a sublist is made only after its copy is finished. To fix this, the new head would have to be written to the table before the sublist starts to be copied; this becomes easier with a representation that has header nodes.

9.13.4 Searching for an atom

Question. Write the routine `contains(L, x)` that finds whether an atom *x* occurs in the list. The path suggested by the slides: compare at an atom, search recursively in a sublist; return at the first find.

```
bool contains(list_pointer L, char x) {
    for (; L; L = L->link)
        if (L->tag ? contains(L->dlink, x) : L->data == x)
            return true;                // return at the first find
    return false;
}
```

The conditional operator selects one of two questions according to the kind of the element: is *x* in the sublist, or is this atom *x*. If the answer is yes, the search ends at once. In the exam-question list, *r* (one of the deepest atoms) and *d* were found and *q* was not. In the worst case, that is, when the atom is absent, all the elements are walked: *O(n)*.

9.13.5 A growing array

Question. Rewrite the array-based list so that it doubles in size when it fills up. The path suggested by the slides: `realloc(a, 2*cap*sizeof *a)`; average *O(1)* insertion.

The solution was given at the beginning of the notes as the array implementation of the list ADT. `struct list` holds the address of the array on the heap, the item count, and the capacity; `list_insert` doubles the capacity when the array is full. The result of `realloc` is put into a separate variable. That doubling gives amortized *O(1)* cost for insertion at the end has been proved with a geometric sum.

9.14 Summary

- The **list ADT** is a contract of operations: for `list_insert` the precondition is $0 \le i \le n$ and the postcondition is "`x` is at position *i*, the length is $n + 1$". The **opaque type** (`typedef struct list List;`) hides the implementation; when the implementation changes, the using code does not.
- In the **array** implementation, access is *O(1)* and insertion and deletion are *O(n)* because of shifting. When a dynamic array is doubled in size, insertion at the end becomes amortized *O(1)*.
- In the **linked** implementation, walking to a position is *O(i)* and linking is *O(1)*. A **pointer to pointer** (`node_pointer *pp`) reduces insertion at the head and in the middle to one piece of code.
- The elements of a **generalized list** are atoms or sublists. An element has the form *[tag | data/dlink | link]*; `tag` is the tag, and `data` / `dlink` is a **tagged union**. A list may be **shared** or **recursive**.

- The recursive pattern: if an atom, process it; if a sublist, descend; then continue with `link`. `copy`, `equal`, and `print_atoms` run in $\Theta(n)$ with this pattern and make $1 + n + s$ calls. They are correct on lists with no sharing and no recursion.
- **Depth**: the empty list is 0; otherwise 1 + the largest depth of the sublists. The recursive version is short and direct; the iterative version needs a local stack and uses no global variable.
- **Inversion** adds two lines to the three-pointer routine: the sublist is first reversed with `invert(&p->dlink)`. It works in place and makes no `malloc` calls.
- A tree is stored with nested lists whose first element is the root label, and with a one-line **linear form**. `print_atoms` performs a **preorder** traversal on such a list. In a binary tree the linear form loses the left–right information; an L/R mark or an empty-child symbol must be added.
- **Reference counting** frees shared lists safely with `ref` in the header node; but on a cycle the counter never drops to zero. **Garbage collection** (mark and sweep) collects cycles as well. C has no garbage collection.

9.15 Key terms

abstract data type (ADT) list ADT interface implementation header file opaque type incomplete type dynamic array capacity amortized analysis cache pointer to pointer generalized list head tail (of a generalized list) multivariate polynomial union (C) tagged union depth traversal preorder binary tree subtree stack overflow tail recursion header node reference count garbage collection memory leak dangling pointer

---

9.16 Test Yourself

1. What are the length, the depth, the total number of elements, and the number of atoms of the list L = *(a, ((b), c), ())*? How many times is `copy(L)` called?

   Answer

   At the top level there are *a*, *((b), c)*, and *()*; the length is 3. The depth of *(b)* is 1, that of *((b), c)* is 2, and that of L is 3. Elements: 3 at the top level, 2 inside *((b), c)*, 1 inside *(b)*; 6 in total. The atoms are *a*, *b*, *c*: 3. Since the number of sublist elements is $s = 3$, `copy` is called $1 + 6 + 3 = 10$ times.

2. What are the head and tail of the list *((a, b), c)*? What is the tail of *(a)*?

   Answer

   The head is the list *(a, b)*, the first element. The tail is the list *(c)* made of the remaining elements, not the atom *c*. The tail of *(a)* is the empty list *()*.

3. `ll_insert(&B, 0, 5)` is called on the list 4 → 7. How many times does the loop run, what does `pp` point to, and which pointer does `*pp = new_node` change?

   Answer

   Because *i = 0*, the loop never runs and `pp` stays at `&B.head`. `new_node->link = *pp` links 5 to 4; `*pp = new_node` turns `B.head` to 5. The list becomes 5 → 4 → 7 with `n = 3`. No separate branch is needed for insertion at the head.

4. `array_insert(&L, 0, v)` is performed ten times on an empty `ArrayList`. How many shifts are made in total? What if the same ten values were always added at the end?

   Answer

   At the *k*-th insertion the list holds *k − 1* elements and all of them shift: *0 + 1 + … + 9 = 45*. For insertion at the end, since *i = n*, the loop never runs: 0 shifts.

5. *(a, (b))* and *(a, b)* are compared with `equal`. What is the result and how many times is the function called?

   Answer

   In call 1, *a*/*a* are equal and call 2 is made for the rest. In call 2, one element is a T and the other an atom; because the tags differ, the block is skipped and `FALSE` is returned. The result is `FALSE`, with 2 calls in total.

6. What does `invert(&L)` turn the list *L = (a, (b, (c, d)), e)* into? How many `malloc` calls are made?

   Answer

   Every level is reversed: *(c, d)* → *(d, c)*, *(b, (c, d))* → *((d, c), b)*, top level → *(e, ((d, c), b), a)*. The routine only changes links; no `malloc` is made.

7. Inside `invert`, if `list_pointer t = p->dlink; invert(&t);` is written with a new local variable

instead of `invert(&p->dlink)`, what happens for *(a, (b, c))*?

Answer

The inner call inverts the sublist correctly and writes the new head *c* into `t`. But the `dlink` field of the T element is not updated; it still points to *b*. Because the link of *b* is now `NULL`, the list appears as *((b), a)*; *c* becomes unreachable and leaks. The fix is either to add `p->dlink = t` or to pass the address of the field directly.

8. *A = (x)* and *B = (A, A, A)* have been built with header nodes with reference counts. What is the counter of *A* at the start? What is it after `list_free(&B)`, and is *A* freed?

Answer

A is pointed to by the variable `A` and by the three T elements of *B*: the counter is 4. While *B* is being freed, the three T elements lower the counter 4 → 3 → 2 → 1. Because the counter is not zero, *A* is not freed; the variable `A` still points to it.

9. What do `print_atoms` and `print_tree` print for the list *(A, (B, (E, H, (I, J, K)), F), (C, G), D)*?

Answer

`print_atoms` prints the atoms in preorder: `ABEHIJKFCGD`. `print_tree` prints the linear form of the tree: `A(B(E(H,I(J,K)),F),C(G),D)`.

9.17 Further reading

- E. Horowitz, S. Sahni, and S. Anderson-Freed, *Fundamentals of Data Structures in C* (2nd ed.), Chapter 4: the source followed in these notes for generalized lists, the multivariable polynomial representation, and the header node with a reference count.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, Chapter 10 and Chapter 17 (amortized analysis, dynamic tables): describes list implementations and the amortized cost of doubling a dynamic table.
- J. McCarthy, "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I", *Communications of the ACM* 3(4), 1960: the paper that introduced LISP, the `car` / `cdr` operations, and garbage collection.
- R. Johnsonbaugh, *Discrete Mathematics*: gives the mathematical background needed for recursion, induction, and trees.

9.18 References

1. E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2nd ed., Section 4.9 (generalized lists, reference counting).
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, MIT Press, Chapter 10 (list implementations).
3. R. Johnsonbaugh, *Discrete Mathematics*.
4. J. McCarthy, "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I", *Communications of the ACM*, 3(4), 184–195, 1960.

PART III

Trees

Chapters 10–13

Trees

Arrays, linked lists, stacks, and queues are linear structures: every element has at most one predecessor and one successor. A large share of real-world data, however, is hierarchical. A folder contains many folders, a manager has many employees, and an arithmetic operation combines the results of two sub-operations. The structure that holds such relationships naturally is the tree. These notes first define tree terminology on an example. They then define the binary tree and prove its basic results about node counts. Next they build the two memory representations of a binary tree, with an array and with pointer-linked nodes, in C. Most of the notes are devoted to traversal: preorder, inorder, and postorder traversals are written first recursively and then iteratively with a stack, and a queue is used for level-order traversal. The last sections cover short recursive procedures that copy a tree, compare two trees, and mirror a tree, and rebuilding a tree from its traversal sequences. The time and memory cost of every procedure is given with its justification.

LEARNING OUTCOMES

Using the terminology
Show the notions of root, parent, child, sibling, ancestor, descendant, degree, level, depth, and leaf on a tree; tell degree and depth apart.

Analyzing binary trees
Prove that the maximum number of nodes on a level is 2^{i-1} and the maximum number of nodes in a tree of depth k is $2^k - 1$; tell perfect, complete, and incomplete trees apart; use the equality $n_0 = n_2 + 1$.

Building the two representations
Represent a binary tree with an array (parent $\lfloor i/2 \rfloor$, children $2i$ and $2i + 1$) and with `tree_pointer` linked nodes; compare the memory cost of the two representations.

Writing traversals
Write preorder, inorder, and postorder traversals recursively and iteratively with a stack; traverse level by level with a queue; find the output of a traversal by hand.

Building tree procedures
Write recursive procedures that print leaf nodes, copy a tree, check whether two trees are equivalent, and swap left and right children; delete a tree without leaving a memory leak.

Computing the cost
Justify the *O(n)* time and the *O(h)* memory, proportional to the depth, of each procedure; state what these bounds are for balanced and skewed trees.

10.1 The tree data structure

In a linked list each node points to the next node. What would happen if a node could point to more than one next node? The list branches, and a tree emerges. This section defines the tree, lists where it is used, and shows the property that sets it apart from the more general structure, the graph.

10.1.1 Definition and uses

Everyday life has many tree-shaped arrangements. In a family tree, each person has their children below them. In an organization chart, below the general manager come the managers, and below them the supervisors. The file system of a computer is also a tree: inside the root folder there are folders, and inside them other folders and files. Every file has a single parent folder. What these arrangements have in common is a single starting point at the top, and every element hanging from exactly one element above it.

In computer science the tree underlies search and sorting structures, data compression, and compilers. When a compiler reads a program, it first parses the syntax and builds the structure of the program as a tree. Code optimization and intermediate code generation work on this tree. Arithmetic expressions, decision processes, move options in games, and database indexes are also kept in trees.

Definition 10.1

A **tree** is a finite set of one or more nodes. One of the nodes is designated the **root**. The remaining nodes are partitioned into $n \geq 0$ disjoint sets $T_1, \ldots, T_n$. Each of these sets is again a tree and is called a **subtree** of the root.

The definition is recursive: it defines the tree in terms of parts that are themselves trees. When the root is removed, a few smaller trees remain; when their roots are removed, still smaller trees remain. This process goes down to trees with a single node. This form of the definition explains why procedures that work on trees are naturally written recursively. A procedure does a small piece of work at the root and then calls itself for each of the subtrees. Since the subtrees are disjoint, no node is processed twice. All the traversal, copy, and comparison procedures in these notes follow this pattern.

10.1.2 Trees and graphs

The most general structure made of nodes and the edges connecting them is the **graph**. A tree is a special kind of graph. In a tree every node other than the root has **a single parent**. In a graph, edges can arrive at a node from several nodes. Figure 10.1 shows the two structures side by side. In the structure on the left, B and C are below A, and D and E are below C; every node receives a single edge from above. In the structure on the right, D receives edges from both B and C. Because D has two parents, this structure is not a tree.

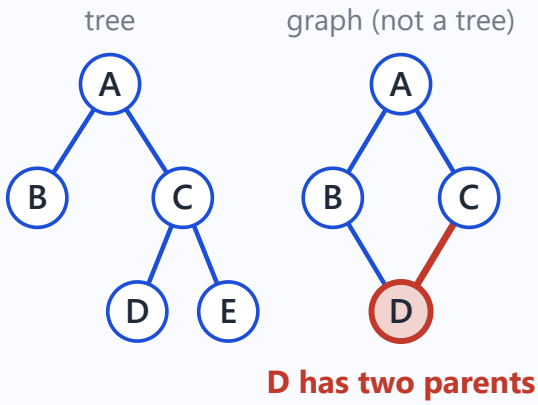


Figure 10.1: On the left, a tree with five nodes; on the right, a graph that is not a tree. In the structure on the right, D can be reached by two different paths; D has two parents.

The single-parent condition has two important consequences. First, the path from the root to any node is unique. Second, a tree has no cycles: starting from a node and following edges, one cannot return to the same node. In a tree with n nodes, exactly one edge enters each node other than the root, so the number of edges is $n - 1$. The relations on the number of nodes of binary trees also rest on this count.

Common mistake

Applying tree procedures to a graph. In a tree a node is reached by a single path, so no visited mark is needed. In a graph, if no mark is kept, nodes are processed again and again; if there is a cycle, the procedure never ends.

10.2 Terminology

Most tree terms come from the family tree: parent, child, sibling, ancestor, grandchild. The remaining terms (degree, level, depth) describe nodes and the tree numerically. This section defines all the terms on a single example tree. Figure 10.2 shows this tree: thirteen nodes, four levels. The small

numbers at the upper right of the nodes are the degrees of the nodes.

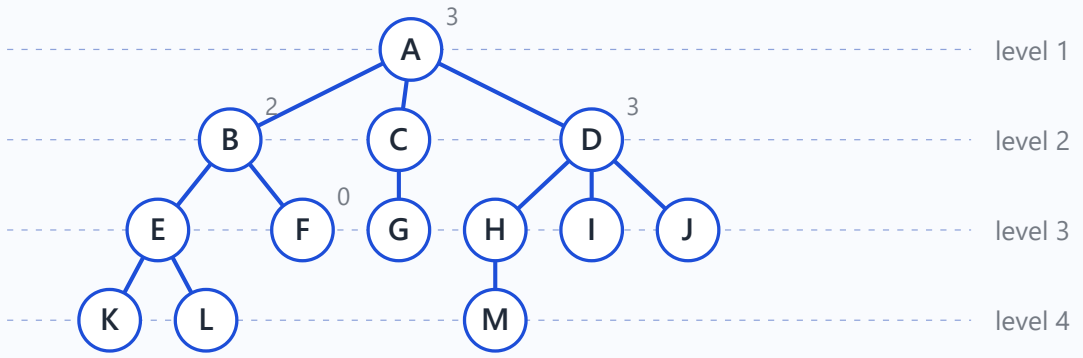


Figure 10.2: A general tree with thirteen nodes, labeled A–M. The root is A. A has three children (B, C, D), B has two children (E, F), and D has three children (H, I, J). The horizontal lines show the levels: the root is at level 1, and K, L, and M are at level 4.

10.2.1 Parent, child, sibling, ancestor, and descendant

The node at the very top of the tree, with no parent, is the **root**. In the example tree the root is A. A node that has subtrees is the **parent** of the roots of those subtrees. The roots of the subtrees are the **children** of that node. B is the parent of E and F; E and F are the children of B. Children of the same parent are **siblings** of one another: E and F are siblings, and H, I, and J are siblings too. The terminology extends to grandparent and grandchild, as in a family tree. D is the grandparent of M, because the parent of the parent of M, H, is D.

The **ancestors** of a node are all the nodes on the path from the root to that node. The ancestors of M are A, D, and H. The root is not a descendant of any node; the root has no ancestors. All the nodes in the subtree of a node are its **descendants**. The descendants of B are E, F, K, and L. Children of children are grandchildren: E, F, G, H, I, and J are the grandchildren of A. The two definitions are inverses of each other: if X is an ancestor of Y, then Y is a descendant of X. This relation matters in traversal procedures. A recursive procedure that processes a node also processes all of its descendants, and never touches its ancestors.

10.2.2 Degree, leaf, level, and depth

The **degree** of a node is the number of subtrees of that node, that is, its number of children. The degree of A is 3, of B 2, of C 1, and of F 0. A node of degree 0 is called a **leaf** or a **terminal node**. The leaves of the example tree are K, L, F, G, M, I, and J. Nodes with at least one child are **internal nodes**: A, B, C, D, E, and H.

The **level** of a node is numbered starting from the root. In this course the root is at level 1; the level of a node is one more than the level of its parent. B, C, and D are at level 2, E, F, G, H, I, and J at level 3, and K, L, and M at level 4. Two numbers are also defined for the tree itself. The **degree of the tree** is

the largest of the node degrees; in the example tree it is 3 (A and D). The **depth of the tree** is the largest level number among the nodes; in the example tree it is 4. Figure 10.3 shows the two numbers in the same drawing.

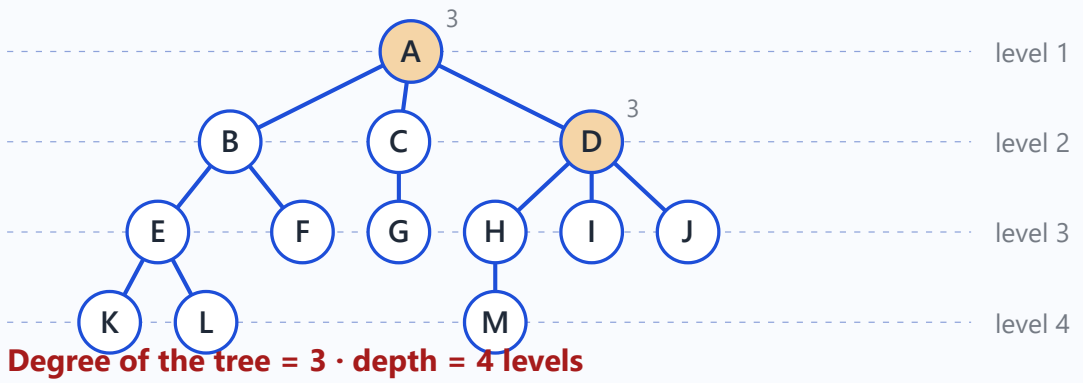


Figure 10.3: The degree of the tree is the degree of the nodes with the largest degree (A and D): 3. The depth of the tree is the largest level number: 4. The two numbers are independent of each other.

Degree and depth answer different questions. Degree measures how *widely* the tree branches, and depth how far *down* it goes. A tree that stretches out like a chain has degree 1 and depth equal to the number of nodes. A tree with a hundred leaves attached directly under the root has degree 100 and depth 2. The memory cost of traversal procedures depends on the depth; in binary trees the degree is at most 2 anyway.

Example 10.1

All the terms for node H in Figure 10.2 are as follows. The parent of H is D; its only child is M. Its siblings are I and J. Its ancestors, starting from the root, are A and D. Its only descendant is M. Its degree is 1, so H is an internal node. Its level is 3: A is at level 1, D at level 2, and H at level 3. The subtree that takes H as the root has two nodes (H and M) and depth 2.

Common mistake

Giving degree and depth with the same definition, as in "the depth of the tree is the largest of the node degrees". This sentence defines the degree of the tree, not its depth. In the example tree the two values are 3 and 4; in a chain they are 1 and n. In exam questions the two are asked separately and cannot be used in place of each other.

10.2.3 Definitions of depth: books count differently

The notions of level and depth shift from source to source. The main source of this course is the book by Horowitz, Sahni, and Anderson-Freed. In this book the root is at level 1 and the depth of the tree is the largest level, that is, the number of *nodes* on the path from the root to the farthest leaf. The books by Cormen et al. (CLRS) and by Sedgewick take the depth of the root to be 0. In those books the **height** of the tree is the number of *edges* on the path from the root to the farthest leaf. For the same tree the two definitions differ by one. Table 10.1 compares the values in the example tree.

Concept	This course (Horowitz–Sahni)	CLRS / Sedgewick
Level of the root	1	0
Level of M	4	3
Depth / height of the tree	4 (depth)	3 (height)
Tree with a single node	1	0
Empty tree	0	−1

Table 10.1: Level and depth in the two source traditions (values for Figure 10.2).

Neither of the two definitions is right and the other wrong; only the thing measured differs. One counts nodes, the other edges. The number of edges on a path is always one less than the number of nodes. So the formulas shift by one too: in this course a binary tree of depth k has at most $2^k - 1$ nodes; written with height, the same bound is $2^{h+1} - 1$. When using a formula in an exam or a source, one should first check which definition applies.

In these notes depth is also used as a function: *depth(NULL) = 0*, and the depth of a tree with a single node is 1. The depth of a non-empty tree is one more than the larger of the depths of its two subtrees. This definition turns directly into a C function:

```
int depth(tree_pointer L) {      /* empty tree 0, single node 1 */
  if (!L) return 0;
  int s = depth(L->left_child);
  int r = depth(L->right_child);
  return 1 + (s > r ? s : r);
}
```

The function first catches the empty tree and returns 0. For a non-empty tree it computes the depths of the two subtrees recursively and adds 1 to the larger. The added 1 is the root itself. Since it visits every node once, its time is $O(n)$. In tests with the binary trees from the slides the function gave the same values as hand computation; for the empty tree it returned 0.

10.3 The binary tree

In a general tree the number of children of a node is unbounded. This flexibility creates a problem in memory: it is not known in advance how many pointer fields to reserve in each node. When the number of children is limited to at most two, both the memory layout becomes fixed and the procedures become simpler. This section defines the binary tree and proves the relations between its number of nodes and its depth.

10.3.1 Definition

Definition 10.2

A **binary tree** is a finite set of nodes. This set is either empty or consists of a root and two disjoint binary trees, called the **left subtree** and the **right subtree**.

Two details of the definition differ from the general tree. First, a binary tree may be empty; in the definition of a general tree there is at least one node. Thanks to the empty tree, the case "there is no left subtree" is expressed as "the left subtree is empty". All procedures also end with a single base case, the empty tree. Second, left and right are distinguished. In a node with one child, it matters whether the child is on the left or on the right. Consider two trees with root A and a single child B: in one, B is on the left; in the other, on the right. As general trees they are the same tree; as binary trees they are two different trees. So node degrees can be 0, 1, or 2, but for a node of degree 1 the direction of the child also carries information.

There is an inclusion relation among these sets: every binary tree (other than the empty one) is a tree, and every tree is a graph. Figure 10.4 shows two binary trees. In the tree on the left, every node has only a left child; this kind of tree is called a **skewed tree**. The tree on the right has four levels and the nodes are distributed evenly over the levels.

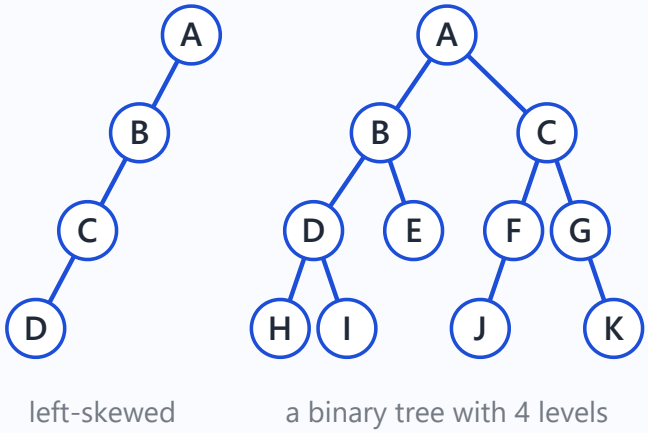


Figure 10.4: Two binary trees. On the left, a left-skewed tree with four nodes: every node has only a left child, depth 4. On the right, a binary tree with eleven nodes and four levels: the depth is again 4, but the number of nodes is almost three times as large.

The two trees in the figure have the same depth; their numbers of nodes are 4 and 11. Very different numbers of nodes can fit in the same depth. Since depth determines the memory cost of traversal procedures, the question "what are the smallest and largest possible depths of a tree with n nodes?" is important. The next two subsections answer it.

10.3.2 Maximum number of nodes on a level and in a tree

In a binary tree every node has at most two children, so the number of nodes at each level is at most twice that of the previous one. This observation gives two bounds.

Theorem 10.1 (Maximum number of nodes per level and per tree)

(i) A binary tree has at most 2^{i-1} nodes on level i ($i \geq 1$). (ii) A binary tree of depth k has at most $2^k - 1$ nodes ($k \geq 1$).

Proof. (i) By induction on i. The base case is $i = 1$: at level 1 there is only the root, and $2^0 = 1$. For the induction step, assume that level $(i - 1)$ has at most 2^{i-2} nodes. Every node at level i is a child of a node at level $(i - 1)$, and every node has at most two children. So level i has at most $2 \cdot 2^{i-2} = 2^{i-1}$ nodes.

(ii) In a tree of depth k the nodes are distributed over levels 1 to k. (i) gives the bound for each level. The total number of nodes is at most

$$\Sigma_{i=1..k}\, 2^{i-1} = 2^0 + 2^1 + \ldots + 2^{k-1} = 2^k - 1$$

The last equality is a geometric sum: writing the sum as S, we get $2S - S = 2^k - 2^0$. ■

The course textbook calls this result a "lemma". A lemma is a small statement whose truth is easy to see and which is used as a step in the proof of a larger theorem. Figure 10.5 shows the counts level by level in a tree of four levels: 1, 2, 4, and 8 nodes, 15 in total. If the depth were 5, 16 more nodes would be added at the fifth level, making 31 in total.

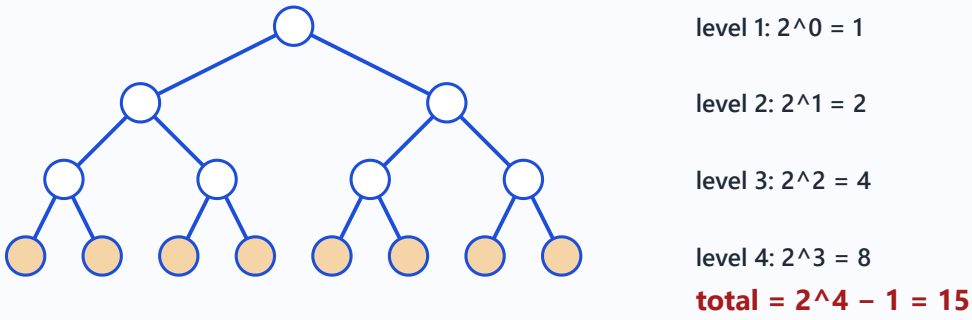


Figure 10.5: A binary tree of depth 4 with all levels full. The numbers of nodes at the levels are 2^0, 2^1, 2^2, and 2^3; the total is $2^4 - 1 = 15$.

10.3.3 Perfect and complete binary trees

Trees that reach the bound of Theorem 10.1 get a special name. There are also trees that come closest to this bound, in which the last level may be incomplete. Because of the similarity of their names, the two notions are often confused.

Definition 10.3

A binary tree of depth k with exactly $2^k - 1$ nodes is called a **perfect binary tree**: all its levels are full. The nodes of a perfect tree are numbered 1, 2, 3, ... starting from the root, from left to right at each level. The n nodes of a binary tree may match one to one the nodes numbered 1 to n of the perfect tree of the same depth. In this case the tree is called a **complete binary tree**.

The numbering rule is simple: if the number of a node is i, the number of its left child is $2i$ and that of its right child is $2i + 1$. The root is 1; the children of the root are 2 and 3; the children of 2 are 4 and 5, and those of 3 are 6 and 7. A complete tree is the tree obtained when these numbers are used from 1 to n without gaps. In everyday words: all levels except the last are full, and the nodes of the last level are arranged from left to right with no gaps.

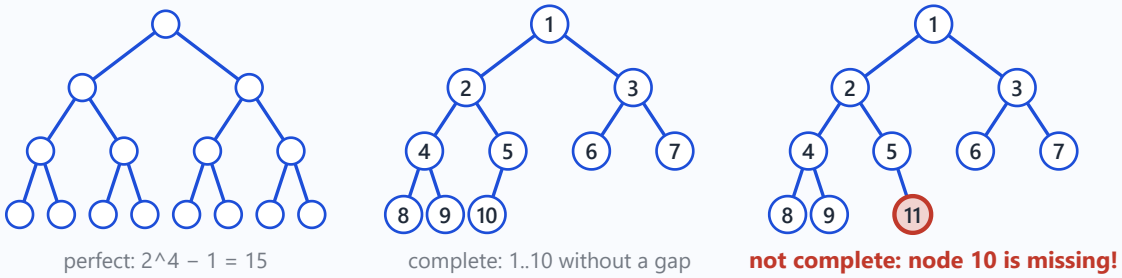


Figure 10.6: On the left, a perfect tree with 15 nodes. In the middle, a complete tree with 10 nodes: the numbers run from 1 to 10 without a break. On the right, an incomplete tree: node number 11 is present and node number 10 is absent.

The tree on the right in Figure 10.6 has ten nodes, but it is not complete. The node numbered 5 has no left child, only a right child. So the numbers are 1–9 and 11; 10 is skipped. In the middle tree, however, 5 has only a left child and the numbers run from 1 to 10 without a break. Every perfect tree is also complete; the converse is not true.

The complete tree matters for two reasons. First, a complete tree fits into an array with no gaps: the node numbered i is put in cell i of the array and no cell is left empty. Second, among binary trees with n nodes, the complete tree has the smallest depth. The heap structure is built on these two properties.

Common mistake

Confusing "complete tree" with "a tree in which every node has either 0 or 2 children". The second notion is called a full binary tree; a tree with all levels full is a perfect binary tree. In the middle tree, node 5 has a single child, yet the tree is complete. Whether a tree is complete is seen from the node numbers: is every number from 1 to n used?

10.3.4 Relations between depth and node count

In a binary tree of depth k, every level has at least one node; otherwise the depth would not be k. So the number of nodes is at least k. By Theorem 10.1 it is at most $2^k - 1$. Written together:

$$k \le n \le 2^k - 1$$

This inequality can also be written in terms of depth. From the right side we get $n + 1 \le 2^k$, that is, $lg(n + 1) \le k$; since k is an integer, $\lceil lg(n + 1) \rceil \le k$. The left side is $k \le n$. As a result

$$\lceil \lg(n + 1) \rceil \le k \le n$$

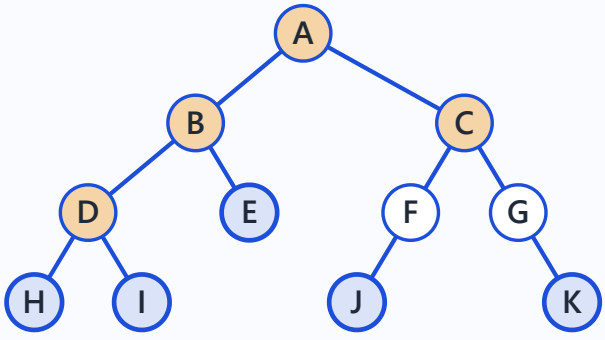
The lower bound is reached by the complete tree: the depth of a complete tree is $\lceil \lg(n+1) \rceil = \lfloor \lg n \rfloor + 1$. The upper bound is reached by the skewed tree: there is a single node at each level and the depth is n. For example, the depth of a binary tree with 20 nodes can be between 5 and 20; the depth of a complete tree with 20 nodes is $\lceil \lg 21 \rceil = 5$. These two extremes are the source of all the cost results of the form "$O(\lg n)$ in a balanced tree, $O(n)$ in a skewed tree".

The second relation on node counts ties the leaves to the nodes with two children. In a binary tree, let the numbers of nodes of degree 0, 1, and 2 be n_0, n_1, and n_2 respectively.

Theorem 10.2 (Number of leaves)

In every non-empty binary tree $n_0 = n_2 + 1$: the number of leaves is one more than the number of nodes with two children.

Proof. The nodes are counted in two ways. Since the degree of every node is 0, 1, or 2, $n = n_0 + n_1 + n_2$. Exactly one edge enters each node other than the root from above; the number of edges is $n - 1$. The edges can also be counted as they leave the nodes: one edge leaves a node of degree 1 and two leave a node of degree 2. Hence $n - 1 = n_1 + 2n_2$. Subtracting the second equation from the first gives $1 = n_0 - n_2$, that is, $n_0 = n_2 + 1$. ■



n = 11: n0 = 5 leaves, n1 = 2, n2 = 4 → 5 = 4 + 1

Figure 10.7: In a tree with eleven nodes, the nodes with two children (A, B, D, C) are shown dark and the leaves (H, I, E, J, K) with an outline. $n_0 = 5$, $n_1 = 2$, $n_2 = 4$, and $5 = 4 + 1$.

Figure 10.7 checks the equality with an example. The two nodes with one child (F and G) do not affect the result: adding a node with one child to a chain changes neither the number of leaves nor the number of nodes with two children. A new leaf is gained only by adding a second child to an existing node; this operation also increases n_2 by one.

The theorem has an important consequence for the C implementation. In the linked representation each node carries two pointer fields; in n nodes there are $2n$ fields in total. Of these, $n - 1$ point to an edge, that is, to a child. The remaining $2n - (n - 1) = n + 1$ fields are `NULL`. In the eleven-node

example, 10 of the 22 fields are filled and 12 are empty. There are more empty pointers than filled ones. If these empty fields are used to point to the previous and next node in inorder, a **threaded binary tree** is obtained. The same number also gives how many times the recursive traversals are called with `NULL`.

10.4 Representing a binary tree in memory

A binary tree can be kept in memory in two ways. In the first, the nodes are placed in an array according to their numbers; the parent and children are found by index arithmetic. In the second, each node is a separate `struct` and is linked to its children by pointers. This section builds both representations in C code and discusses when to choose which.

10.4.1 Array representation

The numbering of a complete tree directly gives an array layout: the node numbered i is put in the cell with index i of the array. To keep the formulas simple, indices start at 1; cell 0 of the C array is not used. The parent and child indices are computed as follows:

- $parent(i) = \lfloor i/2 \rfloor$ (`parent(i)` in the code). If $i = 1$, the node is the root and has no parent.
- $left_child(i) = 2i$. If $2i > n$, there is no left child.
- $right_child(i) = 2i + 1$. If $2i + 1 > n$, there is no right child.

```
#define MAX_NODES 63   /* 2^k - 1 */
char tree[MAX_NODES + 1];  /* [0] unused */
int  n;                    /* node count */

int parent(int i)     { return i / 2; }
int left_child(int i) { return 2 * i; }
int right_child(int i) { return 2 * i + 1; }
/* has a child?  left_child(i) <= n */
```

`MAX_NODES` is the largest number of nodes the array can hold. To be able to hold a tree of depth k in any shape, $2^k - 1$ cells must be reserved; here 63 is chosen for $k = 6$. The size of the array is `MAX_NODES + 1`, because cell 0 is left empty. `n` holds the current number of nodes. The three functions give their results with a single arithmetic operation: since in C the quotient of two positive integers is rounded down, `i / 2` is already $\lfloor i/2 \rfloor$. Whether a child exists is found by comparing the index with `n`: if `left_child(i) <= n`, the left child exists.

The correctness of the formulas can be seen from the numbering. Let the level of node i start at 2^j. The next level starts at 2^{j+1}, and every node to the left of i gives two children there. The number of the left child is $2^{j+1} + 2(i - 2^j) = 2i$, and that of the right child is $2i + 1$. Both $2i$ and $2i + 1$ give i when divided by two and rounded down.

i	`tree[i]`	`parent(i)`	`left_child(i)`	`right_child(i)`
1	A	— (root)	2 (B)	3 (C)
2	B	1 (A)	4 (D)	5 (E)
3	C	1 (A)	6 (F)	7 (G)
4	D	2 (B)	8 (H)	9 (I)
5	E	2 (B)	10 (J)	— (11 > 10)
6	F	3 (C)	— (12 > 10)	—
7–10	G, H, I, J	3, 4, 4, 5	—	—

Table 10.2: Parent and child indices in a complete tree with ten nodes (the letters A–J in cells 1–10). "—" shows that the node does not exist.

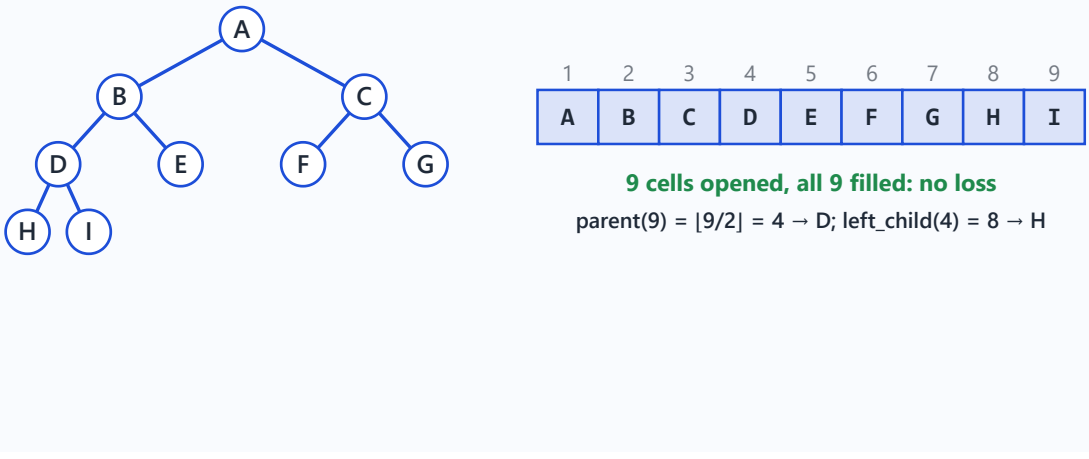
The values in Table 10.2 were produced with a small test program for the code. The left child of E, number 5, is J, number 10; the right child, which would be cell 11, does not exist because 11 exceeds `n` = 10. The nodes numbered 6 and above are leaves. Whether a node is a leaf is also found by the index alone: if $2i > n$, then i is a leaf. In the ten-node tree the leaves are the nodes 6 to 10; in general, in a complete tree the nodes from $\lfloor n/2 \rfloor + 1$ to n are leaves.

Common mistake

Putting the root in cell 0 and using the 1-based formulas. If the root is at 0, `left_child(0)` = 0 gives the root again; the tree folds onto itself. In the 0-based layout the formulas are $2i + 1$, $2i + 2$, and $\lfloor (i - 1)/2 \rfloor$. One of the two layouts must be chosen and the formulas written accordingly; in this course cell 0 is left empty.

10.4.2 Wasted memory in the array

The array representation can be used for any binary tree: the tree is thought of as embedded in the perfect tree of the same depth, and the cells of nonexistent nodes are left empty. But these empty cells waste memory. Figure 10.8 compares two extreme cases.



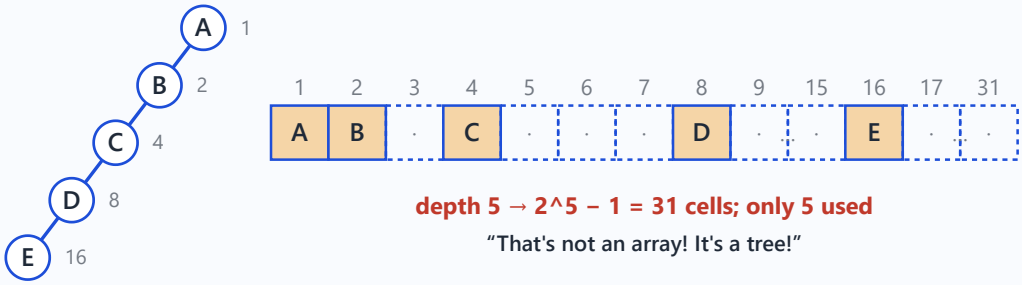


Figure 10.8: Top: a complete tree with nine nodes: nine cells are opened and all nine are filled. For example, *parent(9) = ⌊9/2⌋ = 4* (D) and *left_child(4) = 8* (H). Bottom: a left-skewed tree of depth 5: the nodes fall into cells 1, 2, 4, 8, and 16; only five of the 31 cells are used.

In a complete tree there is either no wasted memory or it is limited to a few cells at the last level. The further the tree is from complete, the more empty cells there are. The worst case is the skewed tree. In a left-skewed tree every node is the left child of the previous one; the numbers double: 1, 2, 4, 8, … A left-skewed tree with k nodes needs $2^k - 1$ cells and uses only k of them. For $k = 5$, 26 of the 31 cells are empty. If $k = 30$, more than a billion cells would be needed; for a list of thirty nodes this is unacceptable. The array representation also causes trouble when the tree changes: inserting or deleting a node in the middle requires moving the whole subtree below it to other indices.

The course instructor sums up the situation like this: storage and access are the two main subjects of computer science; if the tree is a complete binary tree and no insertions or deletions will be made, implementing it with an array is appropriate. In every other case the tree is kept in the linked representation.

10.4.3 Linked representation

In the linked representation each node is a `struct` that carries two pointers along with its data. The left pointer points to the root of the left subtree and the right pointer to the root of the right subtree. If the subtree is empty, the pointer is `NULL`. The definition used in this course is:

```
typedef struct tree *tree_pointer;
struct tree {
    tree_pointer left_child;   /* left subtree */
    char      data;
    tree_pointer right_child;   /* right subtree */
};
```

The first line gives the pointer type pointing to `struct node` the name `tree_pointer`. This line can be written before `struct node` is defined, because C allows defining a pointer to a structure without knowing its contents. Inside the structure there are two fields of type `tree_pointer`; the structure points to objects of its own kind. Such structures are called self-referential structures. The node

structure of the linked list had a single link field; here there are two. The tree itself is represented by the address of the root node: the declaration `tree_pointer root;` is a tree variable. `root == NULL` means the empty tree.

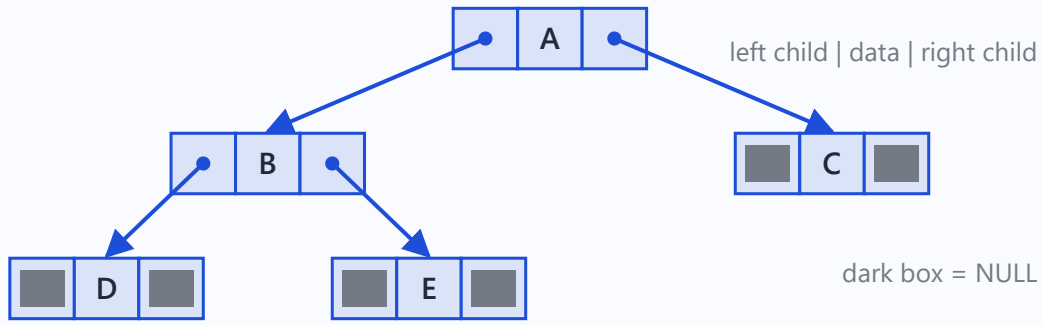


Figure 10.9: Linked representation of the tree A(B(D, E), C). Each node has three cells: left child, data, right child. The small dark boxes are `NULL` pointers. Five nodes have ten pointer fields; four are filled and six ($n + 1$) are `NULL`.

The nodes are allocated one by one from heap memory with `malloc` and may land anywhere in memory. The tree structure is built by pointers alone. Table 10.3 gives one possible memory view of the tree in Figure 10.9. The addresses are chosen as examples; real addresses change on every run. What matters here is that the order of the nodes has nothing to do with the shape of the tree.

Address	`left_child`	`data`	`right_child`
0x1040	NULL	'E'	NULL
0x10a0	0x2020	'B'	0x1040
0x1f00 (`root`)	0x10a0	'A'	0x2060
0x2020	NULL	'D'	NULL
0x2060	NULL	'C'	NULL

Table 10.3: A possible layout in memory of the tree A(B(D, E), C) (the addresses are examples).

The space a node occupies in memory depends on the compiler and the machine. On a 64-bit machine every pointer is 8 bytes and `data` is 1 byte. To align the second pointer to an address that is a multiple of 8, the compiler adds 7 bytes of padding after `data`. In the Windows x86-64 build used for testing, `sizeof(struct node)` came out as 24 bytes: two pointers of 16 bytes, 1 byte of data, and 7 bytes of padding. If the field order were `left_child`, `right_child`, `data`, the padding would move to the end and the size would again be 24. A tree with n nodes takes up the space of n nodes; the shape of the tree does not affect memory. Unlike in the array representation, a skewed tree takes as much space as a balanced one.

10.4.4 Allocating nodes and releasing the tree

For each new node the same three jobs are done: memory is allocated, it is checked that the allocation succeeded, and the child pointers are set to `NULL`. Gathering these jobs in a single function both shortens the code and prevents the errors that a forgotten check would cause. `get_node` from the slides and its inverse `tree_free` are the following:

```
tree_pointer get_node(void) {          /* new node */
    tree_pointer p = malloc(sizeof *p);
    if (p == NULL) {
        fprintf(stderr, "Out of memory!\n");
        exit(EXIT_FAILURE);
    }
    p->left_child = p->right_child = NULL;
    return p;
}

void tree_free(tree_pointer L) {     /* postorder */
    if (!L) return;
    tree_free(L->left_child);
    tree_free(L->right_child);
    free(L);
}
```

The first line of `get_node` asks for room for one node with `malloc(sizeof *p)`. Writing `sizeof *p` gives the size of the type that `p` points to; the expression stays correct even if the type name changes. If `malloc` cannot find space, it returns `NULL`. In that case the function writes a message to `stderr` and ends the program with `exit(EXIT_FAILURE)`. This is enough for homework and exam programs; in a large program `NULL` is returned to the caller and the decision is made there. The memory `malloc` gives is not cleared; the fields hold random values. So the two child pointers are explicitly set to `NULL`. The expression `p->left_child = p->right_child = NULL;` works from right to left: first `NULL` is assigned to `right_child`, and the value of that assignment is assigned to `left_child`. The caller fills in the data field.

`tree_free` returns all the nodes of the tree to the heap. The order matters: first the left subtree and then the right subtree are freed, and the root is released last. If the root were released with `free` first, reading the fields `L->left_child` and `L->right_child` would mean accessing released memory. This is undefined behavior. The read sometimes "works" and sometimes gives garbage; either way, the subtrees either leak or the program crashes. This order, which leaves the root for last, is postorder traversal itself. Since the function visits each node once, it runs in $O(n)$ time. Called with an empty tree it does nothing; `tree_free(NULL)` is safe.

All the C code in these notes was also run with a counting `malloc` / `free` wrapper. At the end of every test the numbers of allocated and released nodes came out equal; no test left a leak. For example, in the traversal tests 53 nodes were allocated and 53 nodes were returned by the `tree_free` calls.

Common mistake

Deleting a tree by releasing only the root node: `free(root);`. This line returns a single node; the addresses of the remaining $n - 1$ nodes were inside the root and can no longer be read. These nodes stay in memory until the program ends. In a long-running program this much memory is lost at every tree deletion. The right way is a procedure such as `tree_free` that visits all the nodes in postorder.

10.4.5 Array or linked representation?

The strengths and weaknesses of the two representations complement each other. Table 10.4 summarizes the comparison.

Criterion	Array	Linked representation
Extra space per node	None (no pointer fields)	Two pointers
Total memory	$2^k - 1$ cells; no loss in a complete tree, very large in a skewed tree	n nodes in every shape
Access to the parent	$O(1)$: $\lfloor i/2 \rfloor$	Not directly (a parent field can be added if needed)
Access to a child	$O(1)$: $2i$, $2i + 1$	$O(1)$: pointer
Insertion and deletion	Subtrees may have to be moved	Only pointer assignments
Where suitable	A complete tree whose shape does not change (e.g., a heap)	Any tree whose shape changes

Table 10.4: Comparison of the array and linked representations.

In short, the rule is this: if the tree is complete and fixed, use an array; for a tree whose shape changes, choose the linked representation. For example, if the computers in a network are fixed and will not be added or removed later, the connection tree can be kept in an array. A heap makes full use of the speed of the array, because it inserts and deletes in a way that always keeps it complete. Search trees, on the other hand, can grow in any direction with insertion; they are kept in the linked representation. The only weakness of the linked representation is access to the parent. If we need to go up from a node, a third pointer, a `parent` field, is added to the node; this costs 8 more bytes per node and one more assignment at every insertion.

10.4.6 Converting a general tree to a binary tree

In a general tree the number of children of a node varies. A has three children while F has none. Reserving, in a fixed-field `struct`, as many pointers per node as "the most children a node can have" leaves most pointers empty. In a tree of n nodes with degree d, only $n - 1$ of the nd fields are filled.

The solution is the **left-child right-sibling** representation: each node carries only two links. The first points to the first (leftmost) child of the node, the second to the sibling immediately to its right.

```
struct gnode {
    char data;
    struct gnode *child;  /* = left */
    struct gnode *sibling;     /* = right */
};
```

This structure has the same form as a binary tree node: `child` takes the place of the left pointer and `sibling` that of the right pointer. So every general tree can be stored as a binary tree, and all the procedures written for binary trees can be applied to it. Figure 10.10 shows the conversion in two steps.

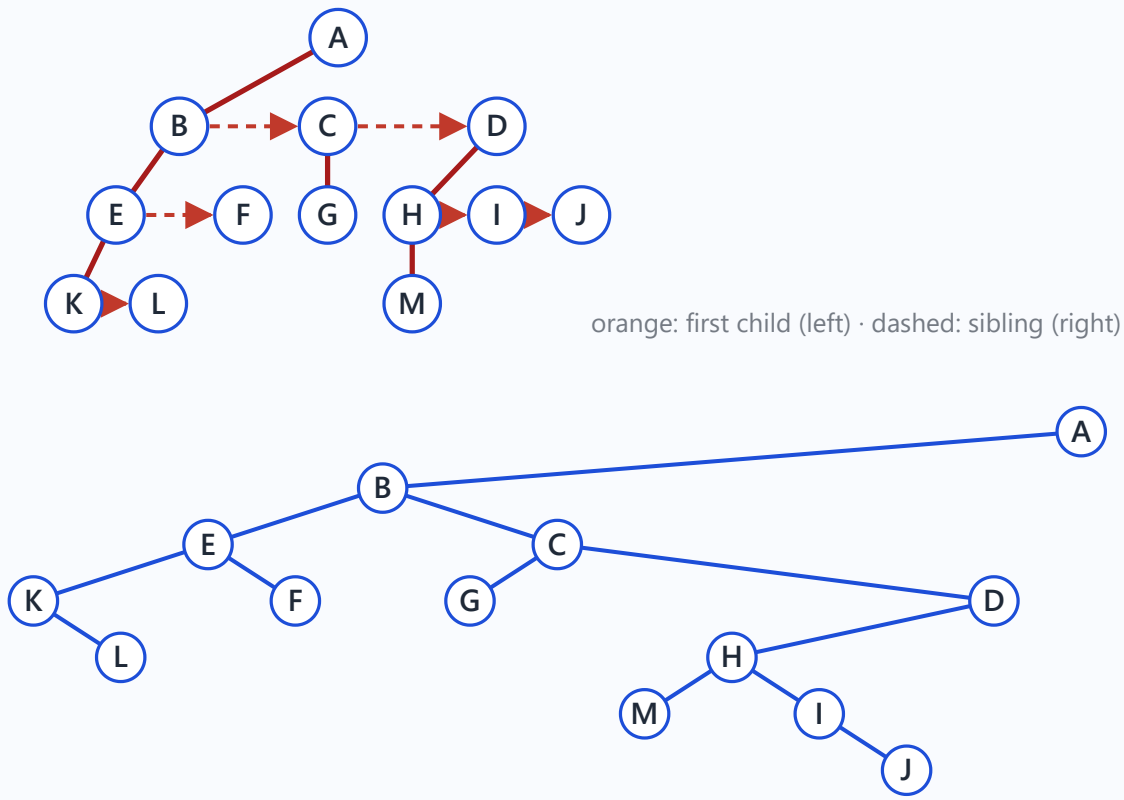


Figure 10.10: Top: in the general tree A–M, only the first-child links (orange) are kept, and the siblings are chained with dashed right-sibling links. Bottom: the same links redrawn as left and right pointers: the result is a binary tree, and the right link of the root is `NULL`.

The conversion is done as follows. For each node the link to the first child is kept and the links to the other children are removed. Then the children of the same parent are tied to one another in a chain from left to right. The first child of B is E; the sibling of E is F. The first child of A is B; the sibling of B is

C, and the sibling of C is D. In the binary tree, B becomes the left child of A, C becomes the right child of B, and D becomes the right child of C. Since the root has no sibling, the right link of the root is always `NULL`.

The conversion has two consequences for traversal. The preorder of the binary tree is the same as the preorder of the general tree: both give A B E K L F C G D H M I J. The inorder of the binary tree equals the postorder of the general tree: K L E F B G C M H I J D A. The reason for the second is this: in the binary tree, the left subtree of a node is all its descendants in the general tree. Inorder traversal first writes the left subtree, that is, all the descendants, and writes the node itself after them. The right subtree is the siblings, and they come after the node. This is the rule "children first, then itself" in the general tree. Both equalities were also checked with a Python script.

10.5 Recursive traversals

Visiting all the elements of a data structure once each in a particular order is called **traversal**. In an array and a linked list the traversal order is natural: from start to end. In a binary tree, two directions are available from each node, and the question "which first?" has more than one answer. This section counts the possible orders, writes three of them in C, traces the code step by step, and derives its cost.

10.5.1 Six traversal orders

At a node there are three things to handle: the data of the node (V), the left subtree (L), and the right subtree (R). A traversal decides in which order to process these three and repeats the same decision in every subtree. Three items have *3! = 6* different orders. Table 10.5 lists them all.

Order	Name
Left – Data – Right (L-V-R)	**inorder**
Left – Right – Data (L-R-V)	**postorder**
Data – Left – Right (V-L-R)	**preorder**
Right – Data – Left (R-V-L)	reverse inorder
Right – Left – Data (R-L-V)	reverse postorder
Data – Right – Left (V-R-L)	reverse preorder

Table 10.5: The six possible orders of processing the data and the two subtrees at a node.

The three orders in which the left is traversed before the right are the standard ones, and they take their names from the position of the root. If the root is processed before the subtrees, it is preorder; if between the two subtrees, inorder; if after both, postorder. The name `preorder` in the C code comes from this term. The remaining three orders are the right-to-left mirrors of the standard orders. If an application requires it, they are used too.

10.5.2 The expression tree and three notations

The names of the traversal orders are familiar from arithmetic expressions. An **expression tree** is a binary tree whose internal nodes are operators and whose leaves are operands. Each operator node is applied to the values of its two subtrees. Figure 10.11 shows the tree of the expression *((((A / B) * C) * D) + E)*. Since the last operator applied is `+`, it is the root; its left subtree is *((A / B) * C) * D* and its right subtree is E.

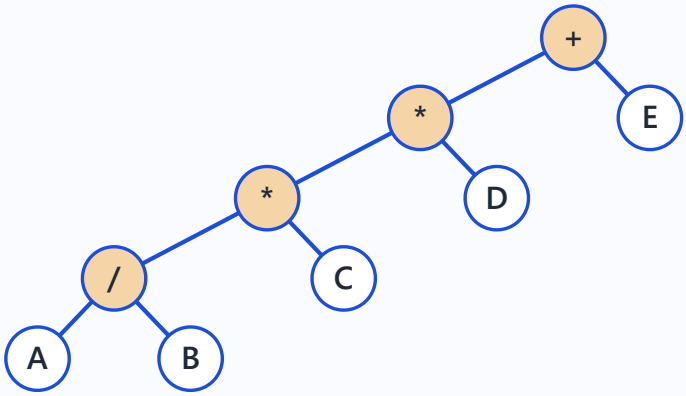


Figure 10.11: The tree of the expression ((((A/B)*C)*D)+E). The internal nodes (orange) are operators and the leaves are operands. The parentheses are stored in the shape of the tree: A/B is deepest, so it is computed first.

When this tree is traversed in the three standard orders, the three well-known notations of the expression appear:

- Inorder: `A / B * C * D + E`. This is the **infix** notation. The tree has no parentheses; when writing infix, parentheses are added around each subtree and *((((A/B)*C)*D)+E)* is recovered.
- Preorder: `+ * * / A B C D E`. This is the **prefix** notation.
- Postorder: `A B / C * D * E +`. This is the **postfix** notation.

Prefix and postfix notations need no parentheses: the order of the operators determines the order of computation unambiguously. Compilers first convert the expressions in source code into such a tree. Postfix notation is evaluated with a stack in a single left-to-right pass: when an operand arrives it is pushed onto the stack; when an operator arrives, two values are taken from the stack, the operation is performed, and the result is pushed back. The postorder traversal of the tree produces exactly this order.

10.5.3 Three traversals of the same tree

To find the output of a traversal by hand, the rule is repeated in every subtree. On the tree A…L with twelve nodes the three traversals are found as follows. In the tree, the left child of A is B and the right child is F. B has no left child; its right child is C. The children of C are D and E; the children of D are G

and H. E has only a right child, I; the children of I are J and K. F has only a left child, L. Figure 10.12 shows the tree numbered in inorder.

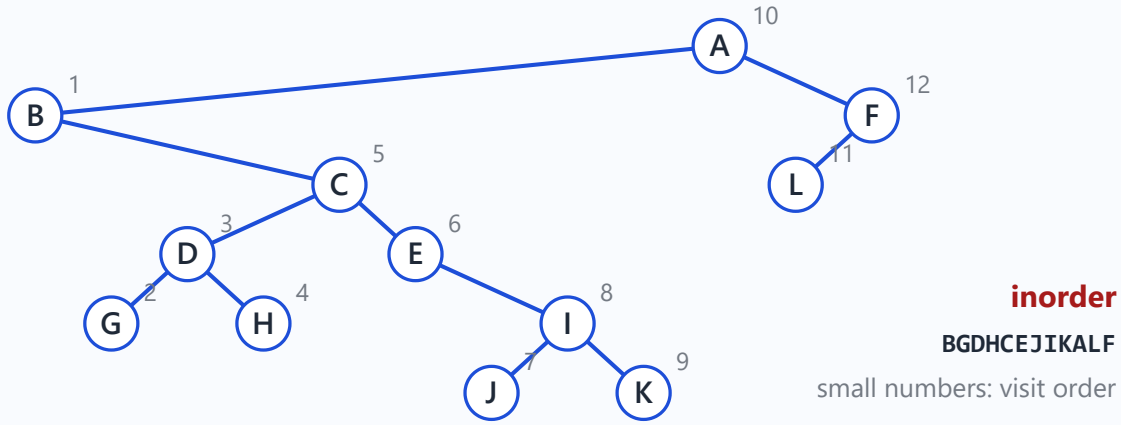


Figure 10.12: Inorder traversal on the tree A...L. Since the left of B is empty, the first node written is B; the output is BGDHCEJIKALF.

- **Preorder (V-L-R).** First the root A is written. Then the whole left subtree: B, followed by the subtree of C, the right child of B (C, D, G, H, E, I, J, K). Last the right subtree: F, L. Output: `ABCDGHEIJKFL`.
- **Inorder (L-V-R).** Start from B, the left subtree of A. Since the left of B is empty, B is written first. Then the right subtree of B is traversed in inorder: G, D, H, C, E, J, I, K. Then A is written, and last the subtree of F: L, F. Output: `BGDHCEJIKALF`.
- **Postorder (L-R-V).** In every subtree the root comes last. G, H, D; then J, K, I, E; then C, B; then L, F; and last A. Output: `GHDJKIECBLFA`.

In the three outputs the relative order of the leaves (G, H, J, K, L) is the same. The traversals differ only in where the internal nodes are placed among the leaves. This observation helps in checking an output found by hand. The first letter of the preorder output and the last letter of the postorder output are always the root.

10.5.4 The traversal procedures

The recursive definition of traversal turns into C code almost word for word. For preorder traversal one says "if the tree is not empty, write the root, traverse the left subtree in preorder, traverse the right subtree in preorder":

```
void preorder(tree_pointer L) {
  if (L) {
    printf("%c", L->data);
    preorder(L->left_child);
    preorder(L->right_child);
  }
}
```

The inorder and postorder procedures have the same skeleton; only the place of the `printf` line changes:

```
void inorder(tree_pointer L) {
  if (L) {
    inorder(L->left_child);
    printf("%c", L->data);
    inorder(L->right_child);
  }
}
```

```
void postorder(tree_pointer L) {
  if (L) {
    postorder(L->left_child);
    postorder(L->right_child);
    printf("%c", L->data);
  }
}
```

The three procedures read line by line as follows. The test `if (L)` is true if `L` points to a non-empty tree. This single test does two jobs. First, a procedure called with an empty tree returns without doing anything; `inorder(NULL)` is safe. Second, it is the stopping condition of the recursion: each call descends into a subtree, the call after a leaf is made with `NULL`, and the chain stops there. `printf("%c", L->data)` writes the data of the node; `L->data` is shorthand for `(*L).data`. The two recursive calls traverse the left and right subtrees by the same rule. If `printf` is before the calls, we get preorder; if between them, inorder; if after, postorder.

The course instructor often warns about these codes: the tree pointer can be `NULL`, and most students do not pay attention to it. The test `if (L)` is therefore mandatory in traversal procedures. If the test is forgotten, the procedure accesses `NULL->data` when it descends to the empty child of the first leaf, and the program crashes. A procedure called with an empty tree crashes on its very first line.

10.5.5 Code trace: inorder traversal

How a recursive procedure works is understood by following the frames on the call stack. Each call adds a frame to the stack; the frame holds the `L` parameter of that call and the return address. When the call finishes, the frame is removed from the stack and execution continues where the caller left off. The trace below is on the four-node tree A(B(D), C): the left child of A is B and the right child is C; B has only a left child, D.

Step	Event	Call stack	Output
1	`L` = A: `inorder(B)` is called for the left	A	
2	`L` = B: `inorder(D)` is called for the left	A B	
3	`L` = D: `inorder(NULL)` is called for the left	A B D	
4	`L == NULL`: `if (L)` is false, the call returns	A B D NULL	
5	`L` = D: its left is done, D is written	A B D	D
6–7	The `NULL` call for the right of D is made and returns at once; the frame of D is removed	A B D NULL	D
8	`L` = B: its left is done, B is written	A B	DB
9–10	The `NULL` call for the right of B; the frame of B is removed	A B NULL	DB
11	`L` = A: its left is done, A is written	A	DBA
12	`L` = A: `inorder(C)` is called for the right	A	DBA
13–14	`L` = C: the `NULL` call for the left is made and returns	A C NULL	DBA
15	`L` = C: C is written	A C	DBAC
16–17	The `NULL` call for the right of C; the frames of C and A are removed in turn	A C NULL	DBAC
18	All calls have returned	(empty)	DBAC

Table 10.6: Trace of `inorder(A)`, tree A(B(D), C). The call stack is written from bottom to top.

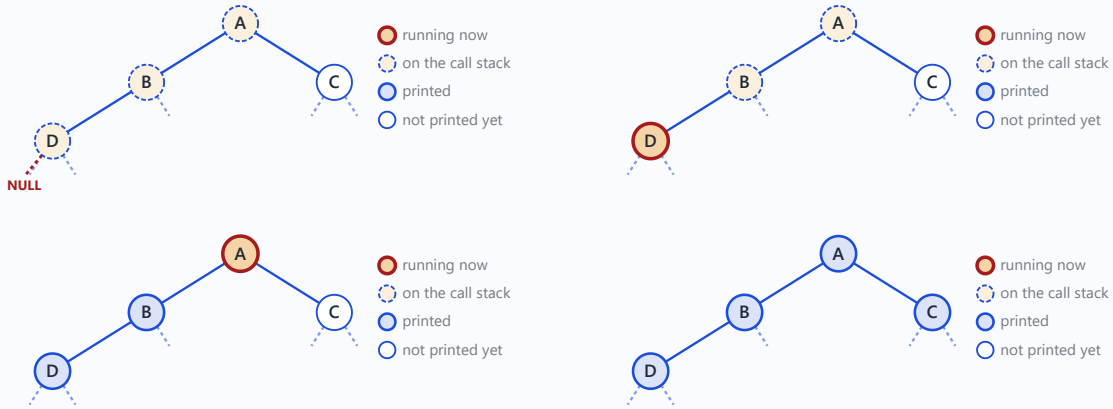


Figure 10.13: Four moments of the inorder traversal: the `NULL` call for the left child of D (step 4), the writing of D (step 5), the writing of A (step 11), and the end of the traversal (step 18). The dashed nodes are frames waiting on the call stack.

Three results come out of the trace. First, the first node written is D: inorder traversal descends first to the leftmost node. Second, at most four frames are on the call stack at the same time: A, B, D, and the frame opened for the empty child of D. This number is one more than the depth of the tree (3). Third, each node is entered once and left once; in the tree with four nodes, five `NULL` calls are made in addition to the node calls. On the same tree, preorder gives `ABDC` and postorder gives `DBCA`. The steps of the trace and the call counts are the same; only the moment of printing changes. In preorder a node is written as soon as its frame opens, and in postorder just before its frame closes.

10.5.6 Cost and NULL calls

Time. The traversal procedure is called once for each node and does a constant amount of work in that call: one test, one `printf`, two calls. In addition there are `NULL` calls for the empty subtrees. The number of these calls is known exactly.

> **Theorem 10.3 (Number of calls in a traversal)**
>
> In a binary tree with n nodes, a recursive traversal procedure is called $2n + 1$ times in total. Of these calls, n are made with nodes and $n + 1$ with `NULL`.

Proof. The procedure is called once for each node: the first call is made with the root and the others with the child pointers of the parent. Each node call makes two more calls; so the number of calls other than the first is $2n$, which together with the first makes $2n + 1$. Of the child pointers, $n - 1$ point to a node and $n + 1$ are `NULL` (a consequence of $n_0 = n_2 + 1$). Therefore $n + 1$ calls are made with `NULL`. ■

The total work is proportional to $2n + 1$ calls; the time is $\Theta(n)$. The shape of the tree does not change the time: a balanced tree and a skewed tree make the same number of calls. In the test with the tree A…L, 25 calls were counted for 12 nodes; 13 of them were `NULL` calls.

Memory. The number of frames on the call stack at the same time equals the length of the path from the root to the current node (plus at most one `NULL` frame). This number is at most the depth h of the tree. The memory cost is therefore $O(h)$. In a balanced tree $h = O(\lg n)$, and in a skewed tree $h = n$. A balanced tree with a million nodes is traversed with about twenty frames; a skewed tree with the same number of nodes needs a million frames. In the test, in the program compiled with zig cc on Windows, a left-skewed chain of 100,000 nodes could be traversed recursively; on a chain of 300,000 nodes the call stack overflowed and the program crashed. The same chain was traversed without trouble by the iterative procedure written with a stack. These limits vary with the machine, the compiler, and the stack size.

The `NULL` calls can be removed by checking before the call:

```
/* before: n+1 calls on NULL */
preorder(L->left_child);
preorder(L->right_child);
/* after: check before calling */
if (L->left_child) preorder(L->left_child);
if (L->right_child) preorder(L->right_child);
```

In the second form the procedure is called only for existing children, and the total number of calls drops from $2n + 1$ to n. In the tree A…L, 25 calls dropped to 12. In this form, though, the `if (L)` at the very beginning of the procedure must remain, because the tree itself may be empty. The gain is a constant factor; the asymptotic time is still $\Theta(n)$. Because the code gets a little longer, the first, short form is preferred in most cases.

The course instructor underlines two things here. First, the claim that "recursive algorithms use a lot of memory" is often an excuse for not writing recursive code: in a balanced tree the stack depth is bounded by $\lg n$, which is a very small number. Second, writing the algorithm is half the job; in computer science every algorithm must be analyzed. That the $O(\lg n)$ bound for memory holds only in a balanced tree, and that the general bound is $O(h)$, is part of this analysis too.

Common mistake

Counting the memory of a recursive traversal as $O(\lg n)$ for every tree. This bound is correct only for balanced trees, whose depth stays of order $\lg n$. A search tree built from sorted keys becomes skewed and its traversal needs n frames. The correct statement is "memory is $O(h)$, $\lg n \le h \le n$".

10.5.7 The call tree

Recursive code is learned by writing, not by reading. In the instructor's analogy, watching a weightlifter a hundred times does not teach you to lift that weight. If a procedure is not understood, the tree formed by the calls is drawn on paper.

In a **call tree** each call is a node; the sub-calls a call makes are its children. The call tree of a traversal procedure is the traversed tree itself: there is a call for each node, and the two children of that call are the calls made for the left and right subtrees. The only difference is that the `NULL` calls are added as leaves. So the call tree has n internal nodes and $n + 1$ leaves. This is a binary tree in which every internal node has exactly two children, and it again fits the equality $n_0 = n_2 + 1$. The contents of the call stack at any moment is the path from the root to the current call in the call tree.

10.5.8 Evaluating an expression tree

Finding the value of an expression tree is a postorder traversal. The value of an operator cannot be computed until the values of its two subexpressions are known. First the left subtree is computed, then the right subtree, and last the operator at the root is applied. To hold the tree, the node structure changes a little: a number is stored in the leaves and an operator in the internal nodes.

```
typedef struct expr *expr_pointer;
struct expr {
    char      op;   /* '+', '-', '*', '/' */
    int       value;   /* leaves only */
    expr_pointer left, right;
};
```

The `op` field holds the character `'+'`, `'-'`, `'*'`, or `'/'`. The `value` field is meaningful only in leaves. `left` and `right` point to the subexpressions. Every internal node has exactly two children, because all four operators are binary; leaves have no children. If a unary operator such as unary minus ($-x$) were added, nodes with one child would also be needed. The evaluation procedure is:

```
int eval(expr_pointer t) {
  if (!t->left && !t->right)        /* leaf: a number */
    return t->value;
  int a = eval(t->left);
  int b = eval(t->right);
  switch (t->op) {
    case '+': return a + b;
    case '-': return a - b;
    case '*': return a * b;
    default:  return a / b;
  }
}
```

The first test checks whether the node is a leaf: if both children are `NULL`, the node is a number and `value` is returned. Otherwise the value of the left subexpression is computed into `a` and that of the right subexpression into `b`. The `switch` returns `a + b`, `a - b`, `a * b`, or `a / b` according to the operator. The `default` branch handles division. The procedure must not be called with an empty tree: if `t` is `NULL`, the first line crashes. Division by zero is not checked in the division; integer division also discards the fractional part. Since the procedure visits each node once, its time is *O(n)*, and because of the call stack its memory is *O(h)*.

Table 10.7 traces the evaluation of the expression *(8 / 4) * 3 + 5*. The root of the tree is `+`; its left subtree is *(8/4) * 3* and its right subtree is 5.

Step	Node	Event	Returned
1	`+`	operator: left subexpression first	
2	`*`	operator: left subexpression first	
3	`/`	operator: left subexpression first	
4	8	leaf	8
5–6	`/` → 4	`a` = 8; the right subexpression 4 is a leaf	4
7	`/`	`b` = 4; 8 / 4	2
8–9	`*` → 3	`a` = 2; the right subexpression 3 is a leaf	3
10	`*`	`b` = 3; 2 * 3	6
11–12	`+` → 5	`a` = 6; the right subexpression 5 is a leaf	5
13	`+`	`b` = 5; 6 + 5	11

Table 10.7: Trace of `eval`, *(8 / 4) * 3 + 5*.

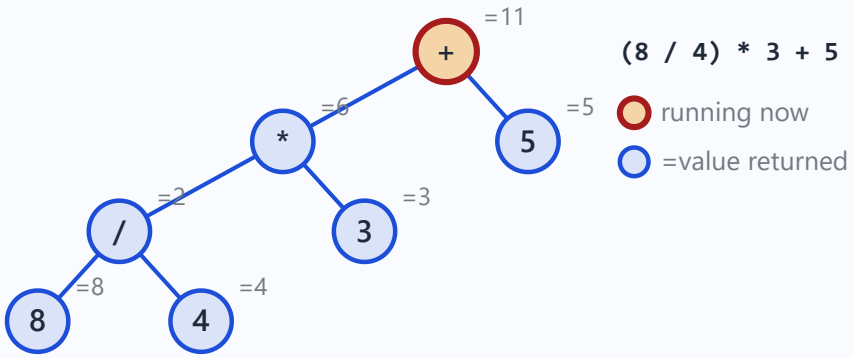


Figure 10.14: When `eval` finishes, the value each node returned is written beside it. The values form from the bottom up, in postorder: 8, 4, 2, 3, 6, 5, 11.

The order in which the values form is 8, 4, 2, 3, 6, 5, 11. This is the postorder traversal of the tree: `8 4 / 3 * 5 +`. The expression tree from the slides was built with a test program and `eval` was called, giving the result 11. With the same program the tree of *(A + B) * (C − D)* was evaluated with A = 7, B = 3, C = 9, D = 4: *10 · 5 = 50* was found.

10.6 Traversal with a stack

Recursive traversal is short and readable, but it falls short in two cases. In very deep trees the call stack can overflow. In some environments (embedded systems, coding standards that forbid recursion) recursion cannot be used at all. This section writes the same traversals with a loop and an explicit stack. It then solves the nine questions in the slides by modifying the same pattern a little: iterative preorder and postorder traversal, printing leaves, and selecting nodes by number of children.

10.6.1 From recursion to a loop

Every recursive algorithm can be made iterative. There are two ways. The first is to express the behavior of the algorithm directly with a loop; for example, the factorial is computed with a `for` loop. The second is to have the program itself do what the operating system does in recursive calls: the places to return to are put on a stack and the loop is fed from this stack. In tree traversal the second way is followed.

Consider inorder traversal. Before a node is written, the whole of its left subtree must be written. Going left from the root, every node passed is held back: first A, then B, then D. When the descent to the left ends, which of the held nodes is written? The one held last, D. After D comes B, after B comes A. So the waiting nodes are reached in the reverse of the order they were added: the last added comes out first. This is exactly the definition of a stack (LIFO). The course instructor stresses this moment with "this is a stack structure!"; it shows that the stack is not added here from outside but arises by itself from the nature of the problem.

What is put on the stack is not copies of the nodes but their addresses. A copy of a node, with its data and two pointers, takes at least 24 bytes; an address is 8 bytes. More importantly, even after it is taken off the stack the copy points to the same subtrees as the original node and is unnecessary: the address is enough to reach the node. At most as many addresses as the leftward depth of the tree are on the stack at the same time.

10.6.2 Inorder traversal with a stack

Two small procedures are enough for the stack. The stack is an array of type `tree_pointer` and an integer holding the top index:

```
void push(int *top, tree_pointer p,
                tree_pointer y[]) {
  if (*top == MAX_SIZE - 1) {
    fprintf(stderr, "Stack is full\n");
    exit(EXIT_FAILURE);
  }
  y[++*top] = p;
}
tree_pointer pop(int *top,
                    tree_pointer y[]) {
  return (*top == -1) ? NULL : y[(*top)--];
}
```

If the stack is full (the top index has reached `MAX_SIZE - 1`), `push` writes a message and ends the program. Otherwise it increases the top index with `++*top` and writes the address into that cell. The expression `++*top` reads as `++(*top)`: it is not the pointer itself but the integer it points to that increases. `pop` returns `NULL` if the stack is empty; otherwise it returns the address on top and decreases the top index with `(*top)--`. Returning `NULL` from an empty stack is very convenient in this implementation, because `NULL` is never pushed onto the stack; a `NULL` return can be used to mean "the stack is empty". The traversal procedure is:

```
void inorder(tree_pointer L) {
  tree_pointer stack[MAX_SIZE];
  int top = -1;
  for (;;) {
    for (; L; L = L->left_child)  // push lefts
      push(&top, L, stack);
    L = pop(&top, stack);  // NULL if empty
    if (!L) break;
    printf("%c", L->data);
    L = L->right_child;
  }
}
```

The code works line by line as follows:

1. `tree_pointer stack[MAX_SIZE]; int top = -1;` opens a local stack. `top` = −1 means an empty stack.
2. The outer loop `for (;;)` is an infinite loop; its only exit is the `break` below.
3. The inner loop `for (; L; L = L->left_child)` starts at `L`, descends to the left as far as it can, and pushes the address of every node it passes. The loop ends when `L` becomes `NULL`: the left child of the leftmost node has been reached.
4. `L = pop(&top, stack);` takes the node on top of the stack. The left subtree of this node is either empty or completely written.

5. `if (!L) break;` : if the stack is empty, the traversal is over.
6. `printf("%c", L->data);` writes the node.
7. `L = L->right_child;` moves to the right subtree of the node. The next iteration of the outer loop again descends to the left in this subtree. If the right subtree is empty, the inner loop does not run at all and the next node is taken from the stack.

The function also works correctly with an empty tree: if `L` is `NULL`, the inner loop does not run, `pop` returns `NULL` because the stack is empty, and the loop ends. If the stack fills up, the program ends with the message "Stack is full". `MAX_SIZE` should be chosen at least as large as the depth of the tree; if the depth is not known in advance, the stack is kept in a growing array.

10.6.3 Code trace

The trace is on the five-node tree A(B(D, E), C): the children of A are B and C, and the children of B are D and E. Table 10.8 gives the contents of the stack and the output at each step. The values were produced with a version of the procedure that prints the stack at every push and pop.

Step	Operation	Stack	`top`	`L` after	Output
1	A is pushed, descend left	A	0	B	
2	B is pushed, descend left	A B	1	D	
3	D is pushed, descend left	A B D	2	NULL	
4	D is popped and written; move right	A B	1	NULL	D
5	B is popped and written; move right	A	0	E	DB
6	E is pushed, descend left	A E	1	NULL	DB
7	E is popped and written; move right	A	0	NULL	DBE
8	A is popped and written; move right	(empty)	−1	C	DBEA
9	C is pushed, descend left	C	0	NULL	DBEA
10	C is popped and written; move right	(empty)	−1	NULL	DBEAC
11	`pop` returns `NULL` : `break`	(empty)	−1	—	DBEAC

Table 10.8: Inorder traversal with a stack, A(B(D, E), C). The stack is written from bottom to top.

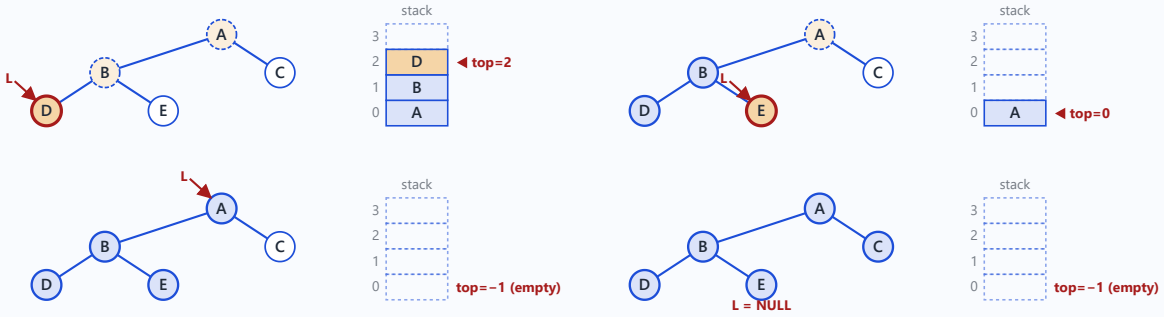


Figure 10.15: Four moments of inorder traversal with a stack: the end of the descent to the left with A, B, D on the stack; the move to the right child E after B has been written; the move to the right child C when A is written; the end of the traversal when the stack empties. The column on the right is the stack array; `top` points to the top cell.

The output is DBEAC and is the same as the output of recursive `inorder`. At most three addresses were on the stack at the same time (step 3): A, B, and D. This number is the number of nodes on the path from the root to the leftmost node. The stack has two roles. On the one hand it takes the place of the call stack of the recursive version. It also marks the moment "its left is done, its own turn is next" for a node: when a node is taken from the stack, its left subtree has been written completely.

10.6.4 Cost and selective printing

Time. Each node is pushed exactly once and popped exactly once. The total number of iterations of the inner loop is n and that of the outer loop is $n + 1$ (the last ends with `break`). The total work is proportional to $2n$ stack operations: $O(n)$.

Memory. At most as many addresses as the depth are on the stack at the same time: $O(h)$. The shape of the tree changes this number markedly. In a left-skewed chain of four nodes, four addresses go onto the stack at once. In a right-skewed chain, on the other hand, each node is popped as soon as it is pushed, because no node has a left child; there is at most one address on the stack and the memory is $O(1)$. In the test, the largest top index for the two chains came out as 3 and 0 respectively.

A nice property of the procedure is that by changing the print line it can be adapted to other jobs. If the `printf` line is tied to a condition, we get a procedure that writes only the selected nodes: leaves, nodes with one child, nodes with two children. When writing the condition one must be careful with the logical operators. The condition `if (!(L->right_child) || !(L->left_child))` means "no right child or no left child". This condition writes leaves and nodes with one child together, that is, nodes with at most one child. For leaves alone we need "no left child *and* no right child": `&&`.

The aim of the course is to be able to write small modules skillfully; that is why small pieces of code are written again and again in different forms. The nine questions in the slides reduce to one of two patterns: recursive or stack-based traversal with a conditional `printf`. If "right to left" is asked for, `left_child` and `right_child` are exchanged.

10.6.5 Iterative preorder traversal

The iterative version of preorder traversal is almost the same as the inorder version. The only difference is the place of printing: the node is written *as it enters* the stack.

```
void iter_preorder(tree_pointer L) {
  tree_pointer stack[MAX_SIZE];
  int top = -1;
  for (;;) {
    for (; L; L = L->left_child) {
      printf("%c", L->data);   // print on entry
      push(&top, L, stack);
    }
    L = pop(&top, stack);
    if (!L) break;
    L = L->right_child;
  }
}
```

In the inner loop that descends to the left, each node is first written and then pushed. So the root is written before its left subtree. A node taken from the stack has already been written; it is used only to move to its right child. The result is the V-L-R order. On the tree A(B(D, E), C) the output is ABDEC.

There is a second solution to the same job. The root is pushed. At each step a node is popped and written; then first the right child and then the left child (if present) are pushed. Since the left child is pushed last, it is popped first; this makes the left subtree be traversed before the right.

```
if (L) push(&top, L, stack);
while ((L = pop(&top, stack))) {
  printf("%c", L->data);
  if (L->right_child)
    push(&top, L->right_child, stack);
  if (L->left_child)
    push(&top, L->left_child, stack);
}
```

This piece is the body of a function; the stack array and `top = -1` are declared before it. The double parentheses in the condition `while ((L = pop(&top, stack)))` tell the compiler that the result of the assignment is deliberately used as the condition; if it were written with single parentheses, `-Wall` would give a warning. Both solutions gave ABCDGHEIJKFL on the tree A...L. In the second solution the nodes waiting on the stack are right children; each node on a path leaves at most one right child on the stack, so the memory is again *O(h)*.

10.6.6 Iterative postorder traversal

Writing postorder traversal iteratively is harder. In inorder traversal, when a node was taken from the stack its left subtree was finished and the node could be written at once. In postorder traversal, a node cannot be written before its right subtree is also finished. A node comes to the top of the stack twice: when we return from the left subtree and when we return from the right subtree. The procedure must tell these two returns apart. For this the last node written is kept in the variable `last`. If the node on top has a right child and the last node written is not that right child, the right subtree has not been traversed yet. First we descend to the right. Otherwise the node is written and removed from the stack.

```
void iter_postorder(tree_pointer L) {
  tree_pointer stack[MAX_SIZE], last = NULL, t;
  int top = -1;
  while (L || top != -1) {
    if (L) {                    // go left
      push(&top, L, stack);
      L = L->left_child;
    } else {
      t = stack[top];           // peek at the top
      if (t->right_child && t->right_child != last)
        L = t->right_child;         // right first
      else {
        printf("%c", t->data);
        last = pop(&top, stack);
      }
    }
  }
}
```

1. The loop runs as long as there is a node to go to (`L`) or a node waiting on the stack.
2. If `L` is not empty, the node is pushed and we descend to the left. This branch corresponds to the call "traverse the left subtree" in the recursive version.
3. If `L` is empty, the descent to the left is over. `t = stack[top]` looks at the node on top but does not take it off the stack.
4. If `t` has a right child and `last` is not that child, the right subtree has not been traversed yet; `L` is set to the right child.
5. Otherwise (there is no right child or it was just written), both subtrees of `t` are finished. `t` is written, popped from the stack, and recorded in `last`.

Why `last` is compared only with the right child is explained as follows. In postorder, the node written immediately before a node is the root of its right subtree if it has one, that is, the right child. If the right subtree is empty, it is the left child, or the node is a leaf. So the question "is the last written node the right child?" tells with certainty whether the right subtree is finished.

Step	Event	Stack	`last`	Output
1	`L` = A is pushed, `L` = B	A	NULL	
2	`L` = B is pushed, `L` = D	A B	NULL	
3	`L` = D is pushed, `L` = NULL	A B D	NULL	
4	Top D: no right child → D is written, popped	A B	D	D
5	Top B: no right child → B is written, popped	A	B	DB
6	Top A: it has right child C and `last` ≠ C → `L` = C	A	B	DB
7	`L` = C is pushed, `L` = NULL	A C	B	DB
8	Top C: no right child → C is written, popped	A	C	DBC
9	Top A: right child C = `last` → A is written, popped	(empty)	A	DBCA
10	`L` is NULL and the stack is empty: the loop ends	(empty)	A	DBCA

Table 10.9: Iterative postorder traversal, A(B(D), C). The stack is written from bottom to top.

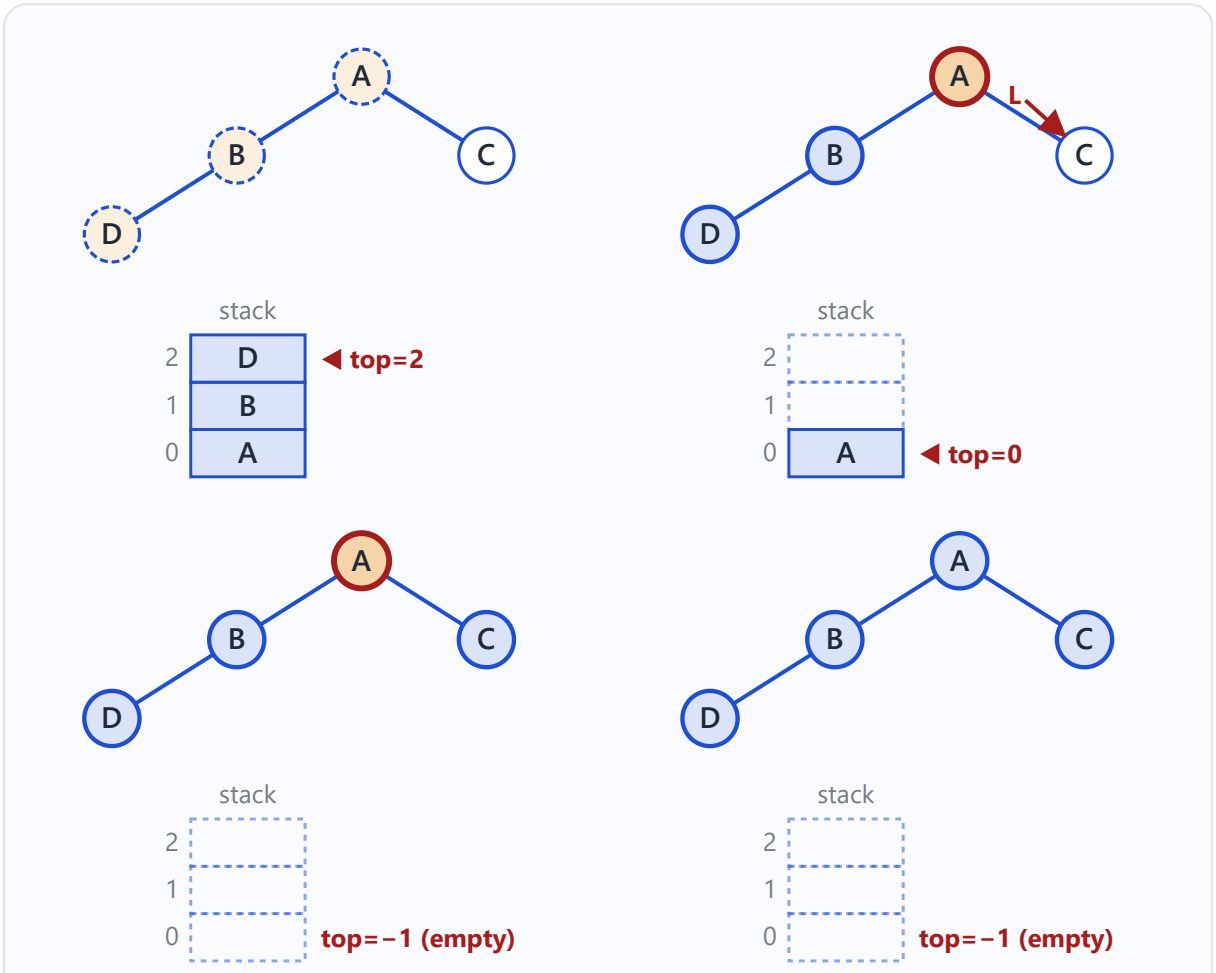


Figure 10.16: Four moments of iterative postorder traversal: A, B, D on the stack (step 3); A on top and the right subtree not yet traversed, descending right (step 6); the right child of A has just been written, A is being written (step 9); the stack is empty (step 10).

A comes to the top twice: at step 6 and at step 9. The first time `last` = B and the right subtree has not been traversed yet; the second time `last` = C and A is written. The **time** is again *O(n)*: each node is pushed once, popped once, and looked at at most twice while on top. The **memory** is *O(h)*. The procedure gave GHDJKIECBLFA on the tree A...L and DEBCA on the tree A(B(D, E), C); both are the same as the recursive version.

Another solution uses two stacks: the V-R-L order is produced with the first stack and pushed onto the second; when the second stack is emptied, L-R-V comes out. This solution is easier to understand but needs *O(n)* extra memory.

> **Common mistake**
>
> Popping the node on top instead of looking at it: `t = pop(&top, stack);` . If the right subtree has not been traversed yet, `t` must be pushed back; if it is not, `t` is never written after the right subtree has been traversed. In postorder traversal a node is taken from the stack only at the moment it is to be written; until then it waits on top.

10.6.7 Printing the leaf nodes

To write the leaf nodes from left to right, any traversal can be used, because the relative order of the leaves is the same in all three standard traversals. In inorder traversal the `printf` is tied to a condition:

```
void print_leaves(tree_pointer L) {
  if (L) {
    print_leaves(L->left_child);
    if (!L->left_child && !L->right_child)
      printf("%c", L->data);
    print_leaves(L->right_child);
  }
}
```

The condition is that the node has neither child. On the tree A...L the output is GHJKL. To write from right to left, only the order of the two recursive calls is changed: first right, then left.

```
void print_leaves_right(tree_pointer L) {        /* Q4 */
  if (L) {
    print_leaves_right(L->right_child);
    if (!L->left_child && !L->right_child)
      printf("%c", L->data);
    print_leaves_right(L->left_child);
  }
}
```

This procedure gives LKJHG on the same tree; it is the reverse of the left-to-right output. The iterative version follows the same path. In the inorder procedure with a stack, the `printf` line is tied to this condition:

```
if (!L->left_child && !L->right_child)
  printf("%c", L->data);
```

To write from right to left iteratively, `left_child` and `right_child` are exchanged everywhere in the stack code: rights are pushed onto the stack, and after a node is popped we go left.

```
void iter_print_leaves_right(tree_pointer L) {      /* Q6 */
  tree_pointer stack[MAX_SIZE];
  int top = -1;
  for (;;) {
    for (; L; L = L->right_child)   // push rights
      push(&top, L, stack);
    L = pop(&top, stack);
    if (!L) break;
    if (!L->left_child && !L->right_child)
      printf("%c", L->data);
    L = L->left_child;
  }
}
```

This code is a reverse inorder traversal (R-V-L). The leaves come out in reverse order. All four procedures were run on the same tree in the test program: the left-to-right versions gave GHJKL and the right-to-left versions gave LKJHG. All run in *O(n)* time; the call stack of the recursive versions and the explicit stack of the iterative versions need *O(h)* memory.

10.6.8 Selecting by number of children

The last three questions select nodes by the state of their children. The recursive procedure that writes the nodes that have only a left child is:

```
void print_left_only(tree_pointer L) {          /* Q7 */
  if (!L) return;
  if (L->left_child && !L->right_child)
    printf("%c", L->data);
  print_left_only(L->left_child);
  print_left_only(L->right_child);
}
```

This procedure is in preorder form: it tests the condition, writes if necessary, and then descends into the two subtrees. The early return `if (!L) return;` at the beginning of the procedure takes the place of the block `if (L) { … }`. The condition `L->left_child && !L->right_child` reads "has a left child and has no right child"; `!L->right_child` means "the right child pointer is `NULL`". The other two questions change only the condition. For nodes that have only a right child, the condition is `!L->left_child && L->right_child`. For nodes with two children, the condition is `L->left_child && L->right_child`:

```
void print_two_children(tree_pointer L) {         /* Q9 */
  if (!L) return;
  if (L->left_child && L->right_child)
    printf("%c", L->data);
  print_two_children(L->left_child);
  print_two_children(L->right_child);
}
```

In the tree A…L, the only node with just a left child is F; the nodes with just a right child are B and E; the nodes with two children are, in preorder, A, C, D, and I. The consistency of the numbers can be checked with Theorem 10.2: the tree has 5 leaves, 3 nodes with one child, and 4 nodes with two children; *5 = 4 + 1* and *5 + 3 + 4 = 12*. All these procedures are examples of the pattern "traverse the tree and write if the condition holds". The kind of traversal determines only the order of the output; the set of selected nodes is independent of the traversal.

Common mistake

Writing `||` instead of `&&` in the leaf condition: `if (!L->left_child || !L->right_child)`. This condition writes all nodes that are missing at least one child. In the tree A…L, B, E, and F are written in addition to the leaves, and the output becomes BGHEJKLF. The compiler does not notice this mistake; it comes to light only when the output is compared by hand with the expected output on a small test tree.

10.7 Level-order traversal

Preorder, inorder, and postorder traversals go into the depths of the tree: once they enter a branch, they do not move to the next branch before finishing it. In some jobs, however, the nodes must be visited by their distance from the root: first the root, then all the nodes at level 2 from left to right, then level 3, and so on. This is called **level-order** traversal. This section solves the same question with two methods and shows why the second method matches a queue naturally.

10.7.1 With level lists

The first method rests on noticing that the order of traversal does not matter. The tree is traversed with any traversal and each node is appended to the end of the list of its own level. When the traversal is finished, the lists are written in order starting from level 1. It is enough to give the traversal procedure the level of the node as a parameter: the root is called with 1, and each child with one more than the level of its parent. For the lists, an array of linked lists is kept; cell *i* of the array points to the head of the list of level *i*.

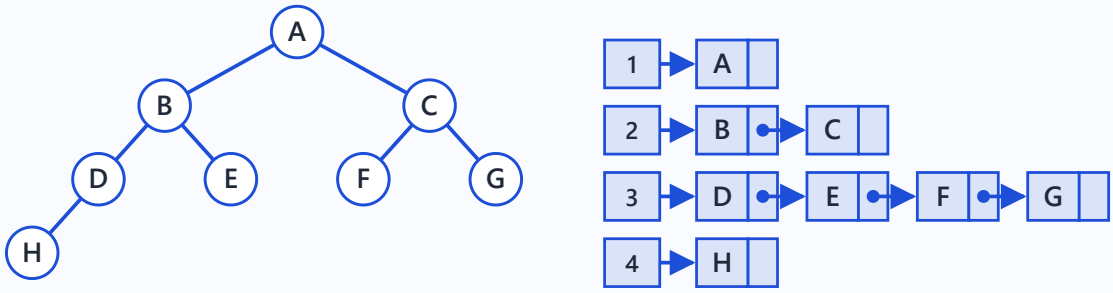


Figure 10.17: On the left, the tree A...H: inorder HDBEAFCG, preorder ABDHECFG, postorder HDEBFGCA. On the right, the level lists built during the traversal: level 1 A, level 2 B C, level 3 D E F G, level 4 H.

Since appending to the lists is always done at the end, the order of the nodes in each list is the order in which the traversal found them. To keep left-to-right order, a traversal that visits the left before the right (preorder, inorder, or postorder) must be chosen. Of two nodes on the same level, the one on the left is found first in all three traversals. In the tree of Figure 10.17 the lists are A; B, C; D, E, F, G; H, and written one after another they give ABCDEFGH.

Cost. Each node is handled three times: it is found in the traversal, appended to a list, and written from the list. Appending to the end of a list is $O(1)$ with a tail pointer; the total time is $3n = O(n)$. The memory is the n nodes of the lists, the h cells of the array of level heads, and the $O(h)$ call stack of the traversal: $O(n + h)$. Since h is at most n, this is $O(n)$.

10.7.2 With a queue

The second method produces the same order without building extra lists. In level-by-level writing, the children of a node must be written after all the other nodes of the same level. When the root is written, its children B and C enter the line. When B is written, the children of B, D and E, also enter the line, but behind C, because C must be written before D and E. The one that enters the line first leaves first: this is the definition of a queue (FIFO). In depth-wise traversal, using a stack came from the nature of the problem; in level-order traversal, using a queue is just as inevitable.

The algorithm is as follows. The root is put into the queue. Until the queue is empty, at each step a node is taken from the front of the queue, written, and its children (first left, then right, if present) are appended to the back of the queue. Since the tree may be empty, the procedure begins with the test

`if (!L) return;`. What is put into the queue is the addresses of the nodes, not copies. The queue is implemented as a circular array:

```
void enqueue(int front, int *rear,
                tree_pointer p, tree_pointer q[]) {
  *rear = (*rear + 1) % MAX_SIZE;
  if (*rear == front) {            /* full */
    fprintf(stderr, "Queue is full\n");
    exit(EXIT_FAILURE);
  }
  q[*rear] = p;
}
tree_pointer dequeue(int *front, int rear,
                     tree_pointer q[]) {
  if (*front == rear) return NULL; /* empty */
  *front = (*front + 1) % MAX_SIZE;
  return q[*front];
}
```

Two indices are kept in the circular queue. `rear` points to the cell of the last element added, and `front` to the cell just before the first element. If the two are equal, the queue is empty. `enqueue` advances `rear` by one; if the end of the array has been reached, it wraps to the beginning with `% MAX_SIZE`. If the advanced `rear` turns out equal to `front`, the queue is full: if one more element were added, an empty queue could not be told from a full one. So in an array of `MAX_SIZE` cells at most `MAX_SIZE - 1` elements are held. `dequeue` returns `NULL` if the queue is empty; otherwise it advances `front` and returns the address in that cell. The index each procedure will change is passed to it by address: `enqueue` changes `rear` and `dequeue` changes `front`. The level-order traversal procedure is:

```
void level_order(tree_pointer L) {
  tree_pointer queue[MAX_SIZE];
  int front = 0, rear = 0;
  if (!L) return;              // tree may be empty
  enqueue(front, &rear, L, queue);
  while ((L = dequeue(&front, rear, queue))) {
    printf("%c", L->data);
    if (L->left_child)
      enqueue(front, &rear, L->left_child, queue);
    if (L->right_child)
      enqueue(front, &rear, L->right_child, queue);
  }
}
```

The line `while ((L = dequeue(&front, rear, queue)))` both takes a node from the queue and tests whether the queue has become empty: when `dequeue` returns `NULL`, the loop ends. The loop body writes the node taken and appends its existing children at the back. `NULL` children are never put into

the queue; so a `NULL` coming out of the queue means only "the queue is empty".

10.7.3 Code trace

Table 10.10 traces the state of the queue on the tree A…H. Cell 0 of the queue array is never used, because the first insertion advances `rear` from 0 to 1. The values were produced with a version of the procedure that prints `front` and `rear` at every removal.

Step	Removed (`front`)	Added	`rear`	Left in the queue	Output
1	A (1)	B, C	3	B C	A
2	B (2)	D, E	5	C D E	AB
3	C (3)	F, G	7	D E F G	ABC
4	D (4)	H	8	E F G H	ABCD
5	E (5)	—	8	F G H	ABCDE
6	F (6)	—	8	G H	ABCDEF
7	G (7)	—	8	H	ABCDEFG
8	H (8)	—	8	(empty)	ABCDEFGH
9	`front == rear` : `NULL`	—	8	(empty)	ABCDEFGH

Table 10.10: Trace of `level_order`, tree A…H. At the start `front` = `rear` = 0; when the root is added, `rear` = 1.

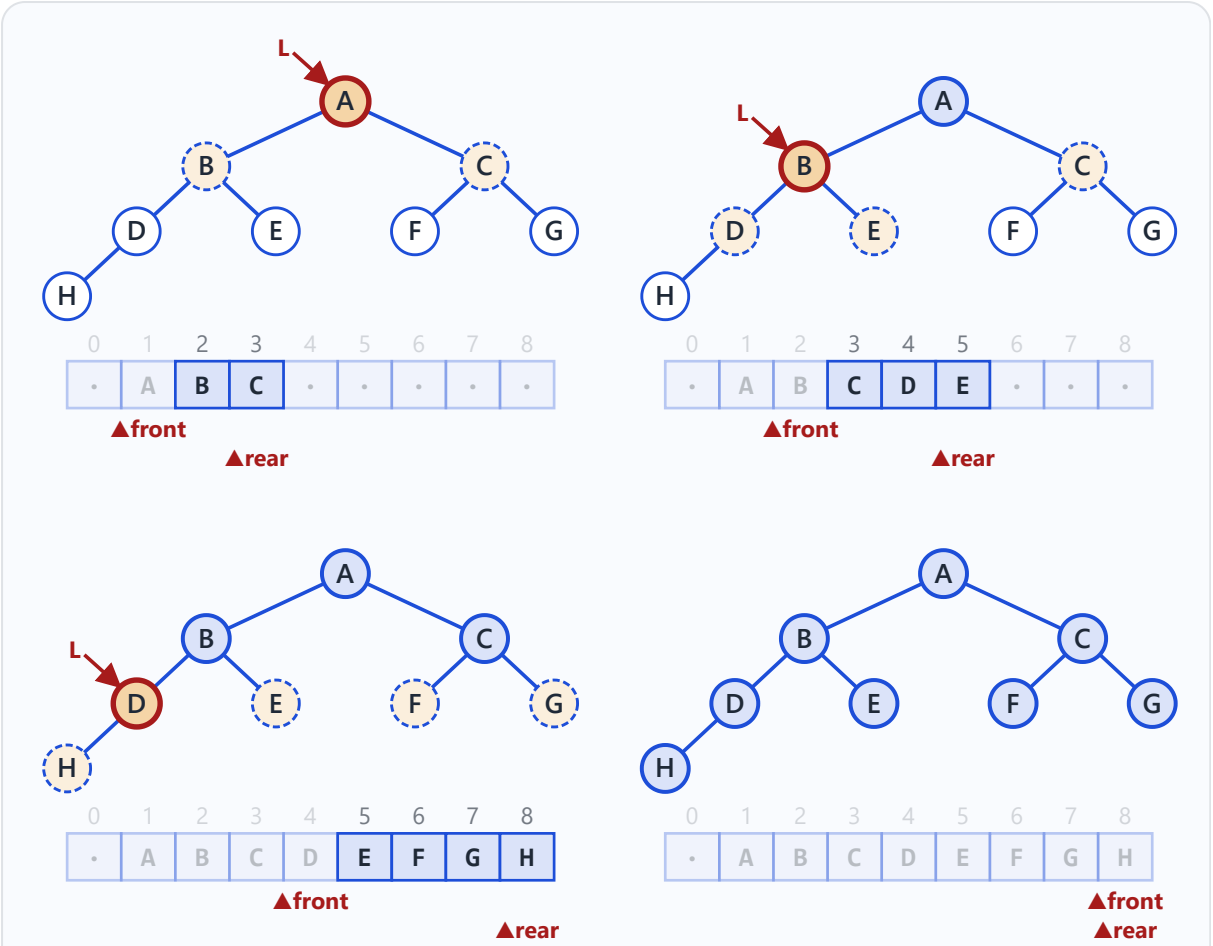


Figure 10.18: Four moments of level-order traversal with a queue: the children B and C of A are in the queue; the children of B have been added too; the child H of D has been added; the queue is empty. The bottom row is the queue array; `front` points to the cell before the first element and `rear` to the last element.

If we look at the contents of the queue, at any moment the queue holds nodes of at most two consecutive levels: the not-yet-written nodes of one level and the children of those ahead of them. After step 4 the queue holds E, F, G from level 3 and H from level 4. The same procedure gave ABFCLDEGHIJK on the tree A…L. When called with an empty tree it returned without writing anything.

10.7.4 Cost: stack versus queue

Time. Each node enters the queue once and leaves once: $2n = O(n)$.

Memory. The queue holds at most as many nodes as the width (w) of one level, plus a few nodes from the next level. So the extra memory is $O(w)$. In a complete tree the last level has about $n/2$ nodes and the memory is $O(n)$. In a skewed tree there is a single node at each level and the queue never holds more than one node: $O(1)$. In depth-wise traversals the situation is the reverse: in a balanced tree the stack needs $O(\lg n)$ and in a skewed tree $O(n)$ memory. Table 10.11 compares the two families of traversal.

Traversal	Auxiliary structure	Extra memory	Balanced tree	Skewed tree
Preorder / inorder / postorder	stack (LIFO)	$O(h)$	$O(\lg n)$	$O(n)$
Level-order	queue (FIFO)	$O(w) \le O(n)$	$O(n)$	$O(1)$

Table 10.11: Auxiliary structure and extra memory of depth-wise and level-order traversal.

The contrast in the table is important: the $O(\lg n)$ memory bound does not hold for level-order traversal with a queue but for depth-wise traversals with a stack in a balanced tree. The same ideas have counterparts in graphs. Level-order traversal with a queue is called **breadth-first search** (BFS) in graphs. When a stack is used instead of a queue, the algorithm turns into **depth-first search** (DFS). Since a node in a graph can be reached by more than one path, visited marks are also kept there.

10.8 Other tree procedures

The traversal pattern applies not only to printing but to almost any job on a tree. The four procedures in this section use the same skeleton: handle the empty tree, do a piece of work at the root, call yourself for the two subtrees. The difference is in what the work at the root is and in how the results of the recursive calls are combined.

10.8.1 Copying a tree

A copy of a tree is a tree with the same shape and the same data but built entirely from new nodes. The copy must not share even a single node with the original tree; otherwise a change made in the copy would also corrupt the original. The procedure is defined recursively as follows: the copy of the empty tree is the empty tree; the copy of a non-empty tree is obtained by attaching the copies of the two subtrees under a new root.

```
tree_pointer copy(tree_pointer L) {
  if (!L) return NULL;
  tree_pointer T = malloc(sizeof *T);
  if (!T) { perror("malloc"); exit(1); }
  T->left_child = copy(L->left_child);
  T->right_child = copy(L->right_child);
  T->data = L->data;
  return T;
}
```

1. `if (!L) return NULL;` returns the copy of the empty tree. This also comes into play in the calls made for the children of leaves.
2. `tree_pointer T = malloc(sizeof *T);` reserves space for the new root. If the allocation fails, `perror` writes an error message and the program ends. Here `malloc` and a check are used directly instead of `get_node`; both forms are correct.

3. `T->left_child = copy(L->left_child);` builds the copy of the left subtree and writes it into the left link of the new root.
4. `T->right_child = copy(L->right_child);` does the same for the right subtree.
5. `T->data = L->data;` copies the data and `return T;` gives the address of the new root to the caller.

The data is written after the two children are copied; the procedure traverses the tree in postorder form. The copy is built from the bottom up: first the leaves, then their parents. The course instructor notes that this procedure works bottom-up and could also be written top-down. By changing the places of the lines one easily moves to the preorder or inorder form: for example, if the line `T->data = L->data;` is moved right after the `malloc`, the data is copied before the children. The result does not change. The recommended way to work is, before writing the code, to draw the calls with a small tree on paper as on the board and to see how the procedure is resolved recursively.



Figure 10.19: Copying the tree A(B, C). On the left, the copy of B is complete and is about to be written into the left link of the copy of A; the data of the copy of A has not been written yet (question mark). On the right the job is done: the copy has the same shape as the original tree but consists of separate nodes.

Copying the tree A(B, C) finishes in sixteen steps: three `malloc` calls for three nodes, four `NULL` calls, three data copies, and writing the returned addresses into the links. In the test program the preorder and inorder outputs of the copy were the same as those of the original tree (ABC and BAC); it was also checked that the node addresses of the copy differ from those in the original tree. The copy of the tree A…L was compared with the original using `equal` and found equal. **Cost:** time *O(n)*, call stack *O(h)*; the copy itself takes up *n* nodes of space. The copy is now an independent tree and must be freed separately with `tree_free` when its job is done.

10.8.2 Equivalence

Two binary trees are equal if they have the same shape and the data at corresponding nodes are the same. If only the shape matters, the trees are called **isomorphic**. The recursive definition is as follows. If both trees are empty, they are equal. If one is empty and the other is not, they are not equal. If both are non-empty, the data at the roots must be equal, the left subtrees must be equal, and the right subtrees must be equal.

```
int equal(tree_pointer L, tree_pointer T) {
  return (!L && !T) ||
         (L && T && L->data == T->data
          && equal(L->left_child, T->left_child)
          && equal(L->right_child, T->right_child));
}
```

The function consists of a single `return` expression. The left side of `||` catches the case where both trees are empty and gives 1. The right side is five conditions joined with `&&`: `L` is non-empty, `T` is non-empty, the data are equal, the left subtrees are equal, the right subtrees are equal. The operators `&&` and `||` of C short-circuit: they do not evaluate the right side once the result is known. This gives two benefits. First, if `L` or `T` is `NULL`, `L->data` is never accessed; the test `L && T` acts as a guard. Second, the recursion stops at the first difference and the remaining nodes are not looked at. If only structural similarity is wanted, the condition `L->data == T->data` is removed.

Step	Call	Event	Result
1	(A, A)	both non-empty, data equal; move to the left subtrees	
2	(B, B)	both non-empty, data equal; move to the left subtrees	
3	(NULL, NULL)	`!L && !T` is true; the right of `||` is not evaluated	1
4	(B, B)	left subtrees equal; move to the right subtrees	
5	(NULL, NULL)	both empty	1
6	(B, B)	all conditions true	1
7	(A, A)	left subtrees equal; move to the right subtrees	
8	(C, D)	data differ; `&&` short-circuits, the subtrees are not looked at	0
9	(A, A)	the right side returned 0	0

Table 10.12: Trace of `equal`, `L` = A(B, C) and `T` = A(B, D).

In the trace five calls are made in total: (A, A), (B, B), two times (NULL, NULL), and (C, D). The children of C and D are never looked at. **Cost.** In the worst case, when the trees are equal, all the nodes of the two trees are compared. If the trees differ, the procedure stops at the first difference; no more comparisons are made than the number of nodes of the smaller tree. The time is $O(min(n_1, n_2))$ and the call stack $O(h)$.

The course instructor draws attention to a contrast here: checking whether binary trees are isomorphic is this easy, whereas finding the isomorphism of two general graphs is a very hard problem. The reason for the difference is the ordered structure of the binary tree. Since the left and right child of each node are known, which node will match which is determined automatically. In a graph the nodes have no order, and the number of possible matchings grows as large as the factorial of the number of nodes.

10.8.3 Writing with early returns

The same procedure can be written, instead of as one long expression, in a form that tests each condition on a separate line and returns at once when the result is known. This way of writing is called guard style (early return):

```
int equal2(tree_pointer L, tree_pointer T) {
  if (L == NULL || T == NULL)
    return L == T;          /* both NULL? */
  if (L->data != T->data)
    return 0;
  if (!equal2(L->left_child, T->left_child))
    return 0;
  return equal2(L->right_child, T->right_child);
}
```

The first test runs when at least one of the two trees is empty. In this case the expression `return L == T;` tells whether both are empty: while one is `NULL`, `L == T` is true only if the other is `NULL` too. The second test compares the data. The third compares the left subtrees and returns 0 at once if they are not equal. The last line leaves the result directly to the comparison of the right subtrees. The behavior is the same as `equal`; in the test the two procedures gave the same results on all pairs of trees.

In both styles the return value of every recursive call is used. This is a point often skipped when writing recursive functions. If the line `equal2(L->left_child, T->left_child);` were written without passing its result anywhere, the computed value would be lost and the function would turn into a test that looks only at the data of the roots. In the instructor's words, a student who says "mine works nicely but it doesn't return" has not yet learned to write recursive algorithms: every call must return a value and the caller must use that value.

> **Common mistake**
>
> Not using the result of a recursive call, or trying to do work after `return`. A procedure of the form `if (L->data != T->data) return 0; equal2(L->left_child, T->left_child); return 1;` compiles and gives the right result when the two trees really are equal. But it also returns 1 for different subtrees, because the result of the recursive call is discarded. The lines after the `return` line never run.

10.8.4 Swapping the left and right children

The mirror image of a tree is obtained by exchanging the left and right subtrees of every node. The procedure swaps not the data but only the pointers:

```
#define SWAP(x, y, t) ((t) = (x), (x) = (y), (y) = (t))
void swap_children(tree_pointer L) {
  tree_pointer p;
  if (L) {
    SWAP(L->left_child, L->right_child, p);
    swap_children(L->left_child);
    swap_children(L->right_child);
  }
}
```

The `SWAP` macro takes three arguments: the two values to be swapped and a temporary variable. `((t) = (x), (x) = (y), (y) = (t))` performs three assignments in a single expression with the comma operator: the value of `x` is saved in `t`, `y` is written into `x`, and the saved value is written into `y`. Using the macro parameters in parentheses makes it expand correctly even with complex arguments such as `L->left_child`. The temporary variable in the procedure is `tree_pointer p`. The root is swapped, and then the procedure is repeated for the two subtrees. Since the swap is done before the calls, the procedure is in preorder form.

After the swap, `L->left_child` is the old right subtree. So on the tree A(B(D), C) the procedure descends first to C and then to B. Table 10.13 gives the steps.

Step	Call	Event
1	A	left (B) and right (C) are swapped; the left of A is now C
2	C	left (NULL) and right (NULL) are swapped
3–4	NULL, NULL	the calls for the two empty children of C return at once
5	B	left (D) and right (NULL) are swapped; the right of B is now D
6	NULL	the new left child of B is empty
7	D	left (NULL) and right (NULL) are swapped
8–9	NULL, NULL	the calls for the two empty children of D return
10	—	done: the tree has become A(C, B(—, D))

Table 10.13: Trace of `swap_children`, A(B(D), C).

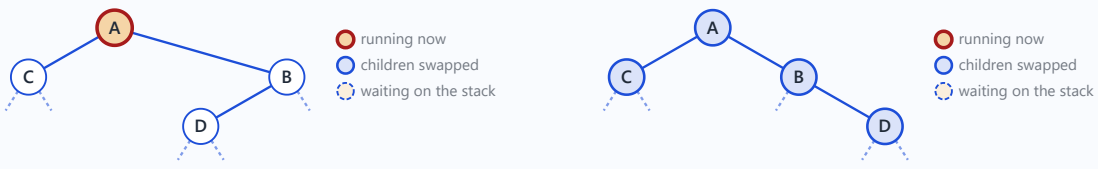


Figure 10.20: On the left, the first step: the children of A have been swapped, C has moved to the left. On the right, the final state: the tree has turned into its mirror image. The inorder sequence has changed from DBAC to CABD, that is, reversed.

As a result the inorder sequence turns from DBAC into CABD. The inorder sequence of a mirror tree is always the reverse of that of the original, because the mirror turns the order L-V-R into R-V-L at every node. The procedure visits each node once and makes a constant number of assignments at each node: time *O(n)*, memory *O(h)*. No node is allocated or freed; the data stay in place and only the pointers change places. If the procedure is applied twice, the tree returns to its original state.

10.8.5 The place of the swap

The `SWAP` line can be placed before, between, or after the two recursive calls. Before and after, the result is the same; in between, the result goes wrong. To see this, a right-skewed chain in which all the subtrees on the right are single-armed is used: B on the right of A, C on the right of B, D on the right of C. Figure 10.21 shows the cases.

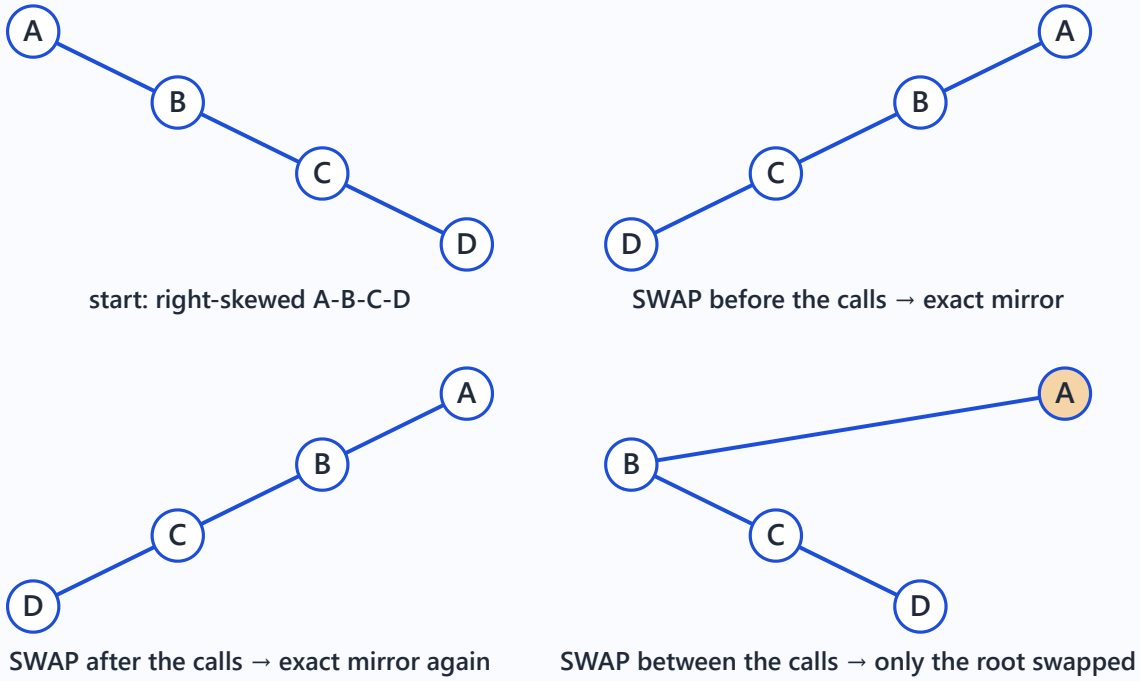


Figure 10.21: The right-skewed chain A-B-C-D (top left) and the results of the three positions of the swap. If the swap is done before or after the calls, the result is the exact left-skewed mirror. If the swap is done between the two calls, only the children of the root change places; the subtree of B is never touched.

- **Swap first (preorder):** At A, left and right are exchanged; B is now on the left of A and the procedure descends into B. The swap also happens at B and C moves to the left. The result is a left-skewed chain: the exact mirror.
- **Swap last (postorder):** First the two subtrees are mirrored internally, then the swap is made at the root. Since each subtree changes places after it has been mirrored, the result is again the exact mirror.

- **Swap in between (inorder):** At A, first the left subtree is processed; since the left is empty, nothing happens. Then the swap is made and B moves to the left. Next the "right" subtree is processed; but the right is now empty. The subtree of B is never entered. In the end only the root has been swapped.

The three versions were run in the test program. With the swap first and last, the inorder sequence came out DCBA and the two trees were found equal with `equal`. With the swap in between, the inorder sequence came out BCDA. The general lesson is this: every call of the procedure reads the subtree it will process from the pointers as they are at the moment of the call. If the swap changes these pointers between the calls, the second call goes to the wrong subtree.

10.9 Rebuilding a tree from traversals

A single traversal output does not determine a tree. For example, there are two trees with inorder output AB: A the root and B the right child, or B the root and A the left child. When two traversals are given together, in most cases the tree is found uniquely. This is a frequently asked exam question. In the question in the slides the inorder sequence is `C M D A K N B F E G` and the postorder sequence is `D A N F E B K M G C`. The binary tree that gives these sequences is to be drawn.

The method rests on two observations. In the postorder sequence, the root of every subtree is the last-written node of that subtree. In the inorder sequence, the letters to the left of a node form its left subtree and the letters to the right form its right subtree. The postorder sequence is read from the end to the beginning. Each letter is the root of a subtree; the inorder sequence separates the left and right of this letter. In reading from the end, the roots of the right subtree come first and then those of the left subtree, because in postorder the right subtree is immediately before the root.

Step	Read from postorder	Position in the inorder sequence	Placement
1	C (last)	no letters on its left	C is the root; the left subtree is empty, the right subtree is MDAKNBFEG
2	G	no letters on its right	G is the right child of C; the right of G is empty, the left is MDAKNBFE
3	M	no letters on its left in this subtree	M is the left child of G; the left of M is empty
4	K	D, A on its left; N, B, F, E on its right	K is the right child of M
5	B	the root of the right subtree of K	B is the right child of K; N on its left, F, E on its right
6	E	the root of the right subtree of B; F on its left	E is the right child of B
7	F	on the left of E	F is the left child of E
8	N	on the left of B	N is the left child of B
9	A	the root of the left subtree of K, D, A	A is the left child of K

Step	Read from postorder	Position in the inorder sequence	Placement
10	D	on the left of A	D is the left child of A; the tree is complete

Table 10.14: Building the tree from the inorder sequence CMDAKNBFEG and the postorder sequence DANFEBKMGC.

The order in the table is the postorder string read from the end to the beginning: C, G, M, K, B, E, F, N, A, D. When the slides draw the same tree step by step, they place the left subtree of K (D and A) before its right subtree; both orders give the same tree. Figure 10.22 shows an intermediate moment and the result.

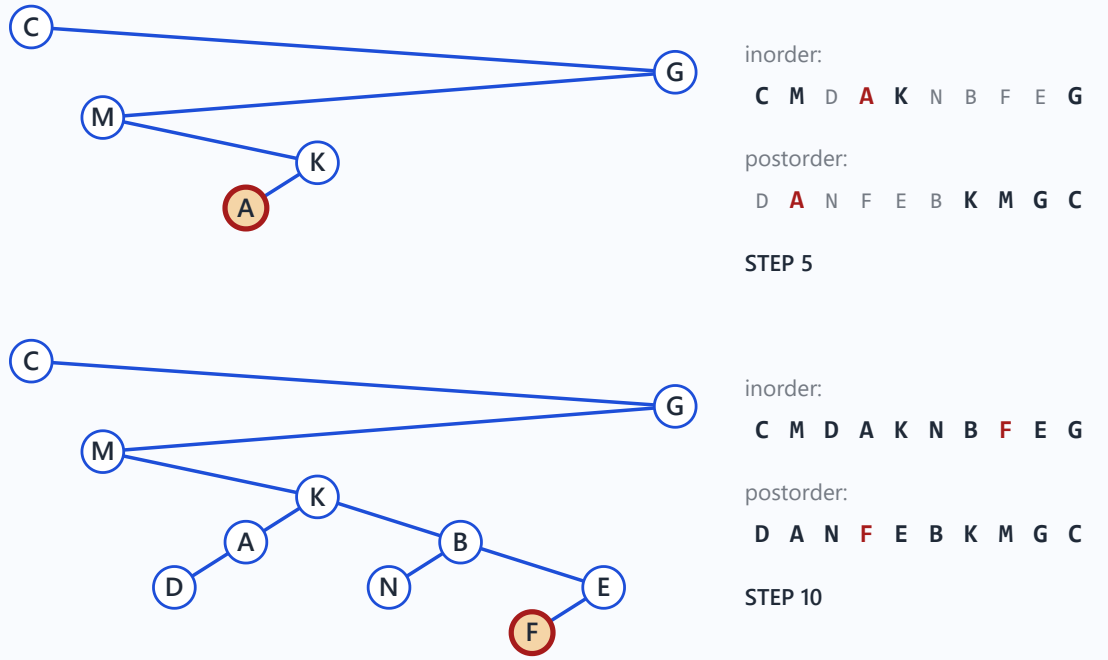


Figure 10.22: Rebuilding the tree. Top: step 5 in the order of the slides: C, G, M, K have been placed and A has been put on the left of K. Bottom: the result, C(—, G(M(—, K(A(D, —), B(N, E(F, —)))), —)). The tree is unique.

The resulting tree was checked with the test program: the inorder traversal of the built tree gave CMDAKNBFEG and the postorder traversal gave DANFEBKMGC. The method turns directly into a recursive C function:

```
/* inord: inorder, postord: postorder string; both n letters */
tree_pointer rebuild(const char *inord, const char *postord, int n) {
  if (n == 0) return NULL;
  tree_pointer T = get_node();
  T->data = postord[n - 1];              /* postorder: root is last */
  int i = 0;
  while (inord[i] != T->data) i++;      /* find the root in inorder */
  T->left_child = rebuild(inord, postord, i);
  T->right_child = rebuild(inord + i + 1, postord + i, n - i - 1);
  return T;
}
```

`inord` is the inorder string and `postord` the postorder string; both hold the `n` letters of the same subtree. The root is the last letter of `postord`. The position `i` of the root is found in the inorder string: the `i` letters to its left are the left subtree and the `n - i - 1` letters to its right are the right subtree. In the postorder string, too, the left subtree takes the first `i` letters and the right subtree the following `n - i - 1` letters. The left subtree is built with `(inord, postord, i)`. The right subtree is built with `(inord + i + 1, postord + i, n - i - 1)`. Pointer arithmetic avoids copying substrings: `inord + i + 1` is the part of the string starting from character `i + 1`. The function assumes that the letters are all distinct and that the two strings belong to the same tree. For each node a linear search is made in the inorder string; in a skewed tree the total time is $O(n^2)$ and in a balanced tree $O(n \lg n)$.

Preorder and inorder sequences also determine the tree uniquely; the only difference is that the root is the *first* letter of the preorder sequence. Preorder and postorder sequences, on the other hand, generally do not determine it. Consider two trees in which A is the root and B the only child: in one B is on the left, in the other on the right. Both have preorder AB and postorder BA. These two traversals find the root and the child, but do not say on which side the single child is. The inorder sequence carries exactly this information: the left subtree is written before the root and the right subtree after it.

10.10 Complexity summary

Table 10.15 collects the time and extra-memory costs of all the procedures in these notes. n is the number of nodes and h the depth; $\lg n \le h \le n$ (balanced ↔ skewed). In this course depth is measured by the number of levels.

Operation	Time	Extra memory	Description
Parent / child (array representation)	$O(1)$	$2^k - 1$ cells	no loss in a complete tree
Recursive preorder / inorder / postorder	$O(n)$	$O(h)$	$2n + 1$ calls, $n + 1$ of them `NULL`
Preorder / inorder / postorder with a stack	$O(n)$	$O(h)$	addresses are pushed on the stack

Operation	Time	Extra memory	Description
Level-order (queue)	$O(n)$	$O(w) \le O(n)$	w: the widest level
Printing leaves, selecting by number of children	$O(n)$	$O(h)$	traversal + condition
`eval` (expression tree)	$O(n)$	$O(h)$	postorder
`copy`	$O(n)$	$O(h) + n$ nodes	postorder
`tree_free`	$O(n)$	$O(h)$	postorder is mandatory
`equal`, `equal2`	$O(min(n_1, n_2))$	$O(h)$	stops at the first difference
`swap_children`	$O(n)$	$O(h)$	pointers only
`rebuild`	$O(n^2)$ worst case	$O(h) + n$ nodes	searching for the root is linear

Table 10.15: Complexity of tree procedures (n nodes, h depth).

Every procedure that touches all the nodes needs $\Omega(n)$ time; all the procedures here reach this lower bound. In memory the deciding factor is the shape of the tree: a balanced tree is traversed with a few dozen frames, a skewed tree with n frames.

10.11 Common mistakes

1. **Accessing `L->data` without a `NULL` check.** The tree pointer can be `NULL`: the tree may be empty, or the recursion may have descended to the empty child of a leaf. The first question of every tree procedure should be "is the tree empty?".
2. **Starting at 0 in the array representation and using the 1-based formulas.** The formulas $2i$ and $2i + 1$ are correct when the root is in cell 1. If the root is in cell 0, the formulas are $2i + 1$, $2i + 2$, and $\lfloor (i - 1)/2 \rfloor$.
3. **Putting a copy of a node on the stack or queue.** The address is enough to reach the node. A copy both takes more space and means an unnecessary memory allocation or structure copy.
4. **Not using the return value of a recursive call, or trying to do work after `return`.** The result of every recursive call that returns a value must either be returned or stored in a variable.
5. **Writing `||` instead of `&&` in the leaf condition.** "Has no children" means that neither child exists. With `||`, nodes with one child are written too.
6. **Mixing up degree and depth, or the definitions of level 1 and depth 0.** Degree measures width, depth measures length. The level of the root is 1 in this course; in other sources it may be 0, and the formulas shift by one accordingly.
7. **Freeing the root first when deleting a tree.** Reading `L->left_child` after `free(L)` is undefined behavior. Deletion must be done in postorder.

10.12 Exercises and solutions

The six questions on the exercise slide of the slides are given below with their solutions. The solutions that contain code were compiled and run with small test programs; the solutions that ask for a tree drawing were checked with a Python script.

10.12.1 The leaf-count equality

Question. Prove the equality $n_0 = n_2 + 1$.

Solution. If the nodes are counted by degree, $n = n_0 + n_1 + n_2$. An edge enters every node other than the root from above; the number of edges is $n - 1$. If the same edges are counted by the nodes they leave, we get $n - 1 = n_1 + 2n_2$. Subtracting the two equations side by side leaves $1 = n_0 - n_2$. The details of the proof are given in Theorem 10.2. In the tree A...L, $n_0 = 5$, $n_1 = 3$, and $n_2 = 4$ were counted; the equality holds.

10.12.2 Three traversals of an expression tree

Question. Draw the tree of the expression *(A + B) * (C − D)* and write the three traversals.

Solution. The last operator applied is `*`; it is the root. The root of the left subtree is `+`, with children A and B. The root of the right subtree is `-`, with children C and D. Preorder gives `* + A B - C D`. Inorder gives `A + B * C - D` and postorder gives `A B + C D - *`. In the inorder output the parentheses are lost; to read the expression correctly, each subtree must be put in parentheses. The prefix and postfix outputs are unambiguous even without parentheses.

10.12.3 Number of nodes

Question. Write the procedure `int count_nodes(tree_pointer L)` that returns the number of nodes of a tree.

```
int count_nodes(tree_pointer L) {           /* node count */
  return L ? 1 + count_nodes(L->left_child) + count_nodes(L->right_child) : 0;
}
```

Solution. The empty tree has no nodes: 0. In a non-empty tree, 1 for the root is added to the two recursive calls for the nodes of the two subtrees. The conditional operator (`? :`) combines the two cases in a single `return` expression. The procedure is in postorder form: the value for the root is computed after the values for the subtrees are known. The time is *O(n)* and the memory *O(h)*. In the test program it returned 12 for the tree A...L, 4 for A(B(D), C), and 0 for the empty tree. With the same pattern, the number of leaves (1 if `L` is a leaf, otherwise the sum for the two subtrees) and the depth are written too.

10.12.4 A tree from the preorder and inorder sequences

Question. Draw the tree whose preorder is `A B D E C F` and whose inorder is `D B E A F C`.

Solution. The first letter of the preorder sequence is the root: A. In the inorder sequence, D, B, E are on the left of A and F, C on the right. The preorder of the left subtree is B, D, E; its root is B. In the inorder sequence, D is on the left of B and E on the right: B(D, E). The preorder of the right subtree is C, F; its root is C. In the inorder sequence, F is on the left of C: C(F, —). The tree is A(B(D, E), C(F, —)). This method is the same as the method of Table 10.14; the only difference is that the root is read from the beginning.

10.12.5 Writing the levels from right to left

Question. Write the queue-based procedure that writes the levels of a tree in order, but each level from right to left.

```
void level_order_right(tree_pointer L) {         /* levels right to left */
  tree_pointer queue[MAX_SIZE];
  int front = 0, rear = 0;
  if (!L) return;
  enqueue(front, &rear, L, queue);
  while ((L = dequeue(&front, rear, queue))) {
    printf("%c", L->data);
    if (L->right_child)
      enqueue(front, &rear, L->right_child, queue);
    if (L->left_child)
      enqueue(front, &rear, L->left_child, queue);
  }
}
```

Solution. Its only difference from `level_order` is the order in which the children are added to the queue: first the right child, then the left. The nodes of the same level enter the queue from right to left and leave in the same order. On the tree A…H the output was ACBGFEDH: level 1 A, level 2 C B, level 3 G F E D, level 4 H. The cost is the same as that of `level_order`.

10.12.6 Are two trees mirror images of each other?

Question. Write the procedure that checks whether two trees are mirror images of each other.

```
int is_mirror(tree_pointer L, tree_pointer T) {
  return (!L && !T) ||
         (L && T && L->data == T->data
          && is_mirror(L->left_child, T->right_child)
          && is_mirror(L->right_child, T->left_child));
}
```

Solution. Its only difference from `equal` is the crossed comparison: the left of `L` is compared with the right of `T`, and the right of `L` with the left of `T`. The tree A(B(D), C) was mirrored with `swap_children` and then compared with the original tree, and it returned 1. Comparing a tree with itself gave 0, because the tree is not symmetric. The time is $O(min(n_1, n_2))$ and the memory $O(h)$.

10.13 Summary

- A tree is a hierarchical structure that starts from a root and splits into disjoint subtrees. Every node other than the root has a single parent; so a tree has no cycles and has $n - 1$ edges in n nodes.
- Basic terms: parent, child, sibling, ancestor, descendant, degree, leaf, internal node, level, and depth. In this course the root is at level 1 and the depth is the largest level; sources such as CLRS count the root as 0.
- A binary tree has at most 2^{i-1} nodes at level i and at most $2^k - 1$ nodes in a tree of depth k. $\lceil \lg(n + 1) \rceil \le k \le n$; the lower bound is reached by the complete tree and the upper bound by the skewed tree.
- $n_0 = n_2 + 1$. In the linked representation, $n + 1$ of the $2n$ pointers are `NULL`.
- A complete and fixed tree is kept in an array (1-based, $\lfloor i/2 \rfloor$, $2i$, $2i + 1$), and a tree whose shape changes with `tree_pointer` links. A skewed tree needs $2^k - 1$ cells in an array.
- A general tree is converted to a binary tree with the left-child right-sibling representation; the preorder is preserved, and the inorder of the binary tree equals the postorder of the general tree.
- Preorder, inorder, and postorder traversals are the same recursive skeleton; they make $2n + 1$ calls and need $O(n)$ time and $O(h)$ memory. An expression tree is evaluated with a postorder traversal.
- Traversals are written iteratively with an explicit stack; node addresses are put on the stack. In the postorder version the last written node is kept.
- Level-order traversal needs a queue; its memory is proportional to the widest level.
- Copying, isomorphism, mirroring, counting, and deleting are examples of the "traverse and process" pattern. Deletion and copying are done in postorder.
- The inorder sequence together with the postorder or preorder sequence determines the tree uniquely; preorder and postorder together do not.

10.14 Key terms

tree root subtree graph parent child sibling ancestor descendant degree leaf internal node level depth height binary tree skewed tree perfect binary tree complete binary tree threaded binary tree traversal preorder inorder postorder level-order expression tree call stack circular queue isomorphic breadth-first search depth-first search balanced tree

10.15 Test Yourself

1. In the general tree A–M (the children of A are B, C, D; the children of B are E, F; the children of E are K, L; the child of C is G; the children of D are H, I, J; the child of H is M), what are the level of H, the degree of D, the ancestors of E, and the descendants of B?

   Answer

   H is at level 3. The degree of D is 3 (H, I, J). The ancestors of E are A and B. The descendants of B are E, F, K, and L.

2. What is the maximum number of nodes in a binary tree of depth 5? What is the maximum number of nodes at level 4 of this tree?

   Answer

   At most $2^5 - 1 = 31$ nodes. At level 4, at most $2^3 = 8$ nodes.

3. A complete binary tree with 20 nodes is kept in a 1-based array. What is the depth of the tree? What are the parent and the children of node 14? Does node 10 have children?

   Answer

   The depth is $\lfloor \lg 21 \rfloor = 5$. The parent of 14 is $\lfloor 14/2 \rfloor = 7$; its children would be 28 and 29, but both exceed 20, so they do not exist (14 is a leaf). Node 10 has a left child, 20; its right child, 21, does not exist.

4. A binary tree has 7 nodes with two children and 3 nodes with one child. How many leaves are there? How many `NULL` pointers are there in the linked representation?

   Answer

   There are $n_0 = n_2 + 1 = 8$ leaves. The total number of nodes is $8 + 3 + 7 = 18$ and the number of `NULL` pointers is $n + 1 = 19$.

5. In a tree, the children of A are B and C and the children of B are D and E; C has only a right child, F. What are the outputs of the four traversals (preorder, inorder, postorder, level-order)?

   Answer

   Preorder ABDECF, inorder DBEACF, postorder DEBFCA, level-order ABCDEF.

6. What is the tree of the expression whose postfix notation is `A B + C *`? What is its prefix

notation?

Answer

The root is `*`; its left subtree is `+` (A, B) and its right child is C. The expression is *(A + B) * C*. The prefix notation is `* + A B C`.

7. Inorder traversal with a stack is applied to a left-skewed chain of 1000 nodes. At most how many addresses are on the stack at the same time? What if the chain were right-skewed?

Answer

In the left-skewed chain, starting from the root, all the nodes are pushed onto the stack while descending to the left: 1000 addresses (`MAX_SIZE` must be at least this large). In the right-skewed chain no node has a left child, so each node is popped as soon as it is pushed: at most 1 address.

8. `equal` is called with the trees A(B, C) and A(B, C(D, —)). What is the result and how many calls are made in total?

Answer

The result is 0. The calls: (A, A), (B, B), two (NULL, NULL) for the two empty children of B, (C, C), and (NULL, D) for the left children of the C's. In the last call both conditions come out false and 0 is returned; since `&&` short-circuits, the right children of the C's are not looked at. Six calls in total.

9. Which tree has preorder ABDGCEF and inorder DGBAECF?

Answer

The root is A. The left subtree is {D, G, B}: its root is B, with D, G on the left of B; D is the root and G is its right child. The right subtree is {E, C, F}: its root is C, left child E, right child F. The tree is A(B(D(—, G), —), C(E, F)).

10. `swap_children` is applied to the tree A(B(D, E), C). What is the inorder sequence of the new tree? What happens if the procedure is applied a second time?

Answer

The original inorder sequence is DBEAC; that of the mirror tree is its reverse: CAEBD. The second application undoes the swap at every node and the tree returns to its original state.

10.16 Further reading

- E. Horowitz, S. Sahni, and S. Anderson-Freed, *Fundamentals of Data Structures in C* (2nd ed.), Chapter 5: the source that these notes follow for terminology, the binary tree lemmas, array and linked representations, traversals, and the tree copy and equality procedures; it continues with threaded binary trees.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms* (3rd ed.), Appendix B.5 and Chapter 12: gives the mathematical definitions of trees (with 0-based height) and traversal in binary search trees.
- R. Sedgewick, *Algorithms in C, Parts 1–4* (3rd ed.), Chapter 5: treats recursion and trees together; contains iterative versions of the traversals with a stack and a queue and proofs of tree properties.
- D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms* (3rd ed.), Sections 2.3–2.3.2: gives binary tree traversal algorithms with a stack and examines in detail the representation of general trees by binary trees.

10.17 References

1. E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2nd ed., Chapter 5 (Trees).
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 3rd ed., MIT Press, Appendix B.5 and Chapter 12.
3. R. Sedgewick, *Algorithms in C, Parts 1–4*, 3rd ed., Addison-Wesley, Chapter 5 (Trees).
4. D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, 3rd ed., Addison-Wesley, 1997, Section 2.3.

Search Trees

In a sorted array, binary search discards half of the remaining range at every step and finds the target in *O(lg n)* steps. But inserting a new element into a sorted array means shifting all the elements behind it by one cell. In a linked list, insertion is cheap, but now the middle element cannot be reached directly and search stays linear. The binary search tree combines the good sides of these two structures. Every node is larger than all the values to its left and smaller than all the values to its right, so each comparison, starting at the root, eliminates one side of the tree. Insertion and deletion change only a few pointers. These notes first define the binary search tree. They then write the insertion, search, depth, and deletion procedures in C and trace each one step by step. The last sections are about balance. We show that sorted input turns a tree into a list; the rotations of the AVL tree, which prevent this, are treated in detail, and the red-black tree and the B-tree more briefly.

LEARNING OUTCOMES

Applying the definition
Check the four properties of a binary search tree on a given tree and explain why inorder traversal produces the values in sorted order.

Insertion and search
Write `bst_insert` and `bst_search` both iteratively and recursively; explain why the root pointer is passed by address when inserting into an empty tree.

Computing the depth
Write the depth function recursively, avoiding global variables, unnecessary extra parameters, and the double-evaluation trap of macros.

Deleting
Delete a node with no children, one child, or two children using a pointer to a pointer; compare the effect of four deletion methods on tree depth.

Keeping balance
Compute the cost of imbalance; recognize the LL, RR, LR, and RL cases in an AVL tree and apply single and double rotations.

Choosing a structure
Tell the binary search tree, the AVL tree, the red-black tree, and the B-tree apart by where each is used.

11.1 The binary search tree

A tree is a hierarchical structure in which every node has a parent and zero or more children. The one node with no parent is the **root**, and the nodes with no children are **leaves**. In a **binary tree** every node has at most two children, distinguished as the left child and the right child. A node together with all the nodes below it forms a **subtree**. In these notes the **depth** of a tree is the number of nodes on the path from the root to the farthest leaf. A tree with a single node has depth 1, and the empty tree has depth 0. Three orders are used to traverse a tree. In **preorder** traversal the root is visited first, then the left and right subtrees. In **inorder** traversal the order is left subtree, root, right subtree. In **postorder** traversal the root comes last. These notions are known from the chapter on trees; they are repeated here only as a reminder.

11.1.1 Why a new structure?

Consider a student records system. Every day it looks up records by student number thousands of times. New students enroll, and graduates leave the system. If the records are kept in an unsorted array, the whole array is scanned for a search, and the cost is $O(n)$. If the records are kept sorted by number, binary search answers in $O(\lg n)$. But now every insertion and deletion shifts elements to keep the order, which means $O(n)$ work. In a linked list, insertion and deletion take a few pointer changes. But one cannot jump directly to the middle of the list; binary search cannot be used, and search is again $O(n)$.

The binary search tree carries the idea of binary search over to a linked structure. In a sorted array, binary search first looks at the middle element. If the target is smaller, it continues in the left half; if larger, in the right half. The root of a binary search tree acts like this "middle element". The left subtree takes the place of the left half and the right subtree the place of the right half. The difference is that the halves are not consecutive cells of an array but nodes linked by pointers. To insert a new element, no element is shifted; only a `NULL` pointer at the right place is changed to point to the new node.

11.1.2 Definition and inorder traversal

Definition 11.1

A **binary search tree** (BST) is a binary tree that may be empty and, if it is not empty, has the following four properties:

1. Every element has a distinct value; the values are unique.
2. All values in a non-empty left subtree are smaller than the value at the root.
3. All values in a non-empty right subtree are larger than the value at the root.
4. The left and right subtrees are also binary search trees.

The fourth property makes the definition recursive: the same condition holds at every node, for the subtree rooted at that node. The word "all" in the second and third properties is important. The condition covers not only the children of a node but also the deepest elements of the subtree. Even the rightmost element at the very bottom of the root's left subtree must be smaller than the root.

The first property uses the tree like a **set**. The same value does not enter the tree twice. In practice the value is often the **key** of a record: a student number, a national ID number, or a word. The other fields of the record sit in the node together with the key. If duplicate keys must be allowed, either a counter is added to the node or equal keys are always placed on the same side by a rule (for example, to the right). In these notes the definition is followed: if an equal value arrives, no insertion is made.

The definition makes the binary search tree a subset of binary trees. Every binary search tree is a binary tree; the traversal, copy, and equality procedures written for trees apply to it as well. The converse is not true. In an ordinary binary tree the values may appear in any order. The most important consequence of the order in the definition is the following theorem.

Theorem 11.1 (Inorder traversal is sorted)

A binary tree is a binary search tree if and only if its inorder traversal lists the values in strictly increasing order.

Proof. First suppose the tree is a binary search tree, and use induction on the number of nodes. For the empty tree and the one-node tree the claim is clear. In a larger tree, inorder traversal writes the left subtree first, then the root, then the right subtree. The left and right subtrees are also binary search trees; by the induction hypothesis each is written in increasing order. All values in the left subtree are smaller than the root and all in the right subtree are larger. So when the three parts are put one after another, the sequence is increasing from beginning to end. For the converse, suppose the inorder sequence is strictly increasing. Every value of the left subtree is written before the root and every value of the right subtree after it. Since the sequence is increasing, the left values are

smaller than the root and the right values larger. The values are also distinct. The inorder sequences of the subtrees are consecutive pieces of this sequence, and they are increasing too. By induction the subtrees are binary search trees as well. ■

The theorem has a practical consequence. To sort a set of values, insert the values one by one into an empty binary search tree, then traverse the tree in inorder. This method is called **tree sort**. If the tree stays balanced, the total cost is $O(n \lg n)$. The theorem is useful in a second way: whether a tree is a binary search tree can be checked in $O(n)$ time by traversing it in inorder and verifying that every value is larger than the previous one.

11.1.3 Checking the definition

Figure 11.1 shows three binary trees. If each node in the left tree is compared only with its own children, no problem appears. The left child of 15 is 9 and the right child is 22, and $9 < 15 < 22$. The children of 20 are 19 and 21; they are in the right places too. But 22 lies in the left subtree of the tree whose root is 18. The second property of the definition requires *all* values in the left subtree to be smaller than 18. 22 violates this condition; the tree is not a binary search tree. If 22 is searched for in this tree, the comparison $22 > 18$ at the root sends the search to the right, and 22 is never found.

The tree in the middle looks unbalanced, but it is valid. Every value to the left of 95 (80, 50, 2) is smaller than 95; the left of 80 is 50, the left of 50 is 2, and the order is broken nowhere. The definition says nothing about the shape of the tree; it only restricts the arrangement of the values. In the tree on the right, every value to the right of the root is larger than 35. To the left of 40 stands 38, and to its right stands 39. Both are smaller than 40 and larger than 35. These values satisfy the condition of every node on the path from the root to them; the tree is valid.

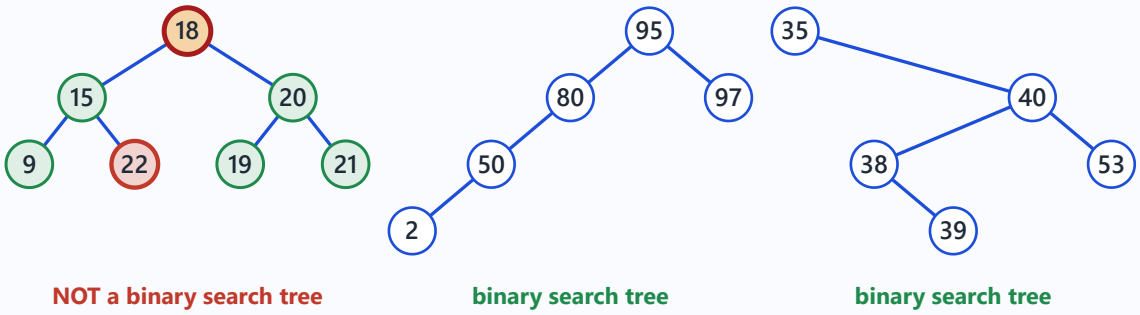


Figure 11.1: Which is a binary search tree? In the left tree, 22 is larger than 15 because it is to the right of 15; but it sits in the left subtree of 18 and is larger than 18 as well, which violates the definition. The middle and right trees are valid: each value satisfies the condition of all the nodes on the path from the root to it.

The examples also show how the check should be done. Every node's value has a lower and an upper bound. The root has no bounds. When we go left from a node, the upper bound becomes the value of that node; when we go right, the lower bound does. In the left tree, 22 is reached from the root by

going left and then right. The lower bound is 15 and the upper bound is 18; 22 does not fall in this range. A C implementation of this idea is given in the exercises section.

> **Common mistake**
>
> Checking a binary search tree by comparing each node only with its children. This check accepts the left tree in Figure 11.1 as valid. A correct check either verifies that inorder traversal is increasing or carries to each node the (lo, hi) range that comes down from the root.

11.1.4 The node structure in C

Each node of the tree is represented by a `struct`. The node holds a data field and two pointers. One pointer points to the left child and the other to the right child. If there is no child, the value of the pointer is `NULL`. The definition from the slides is below.

```
typedef enum { FALSE, TRUE } boolean;
typedef struct node *tree_ptr;
struct node {
    tree_ptr left;
    int       data;
    tree_ptr right;
};
/* get_node(): malloc + NULL check,
sets both children to NULL */
```

The first line builds the logical type, which C does not have, with an enumeration (`enum`): `FALSE` gets the value 0 and `TRUE` gets 1. The second line defines the name `tree_ptr` to mean "pointer to `struct node`". This definition comes before the structure itself. In C it is allowed to declare a pointer to a structure that has not been defined yet; the only thing the compiler needs to know is the size of a pointer. This way a pointer to the structure's own type can be used inside the structure. Such structures are called **self-referential structures**.

On a 64-bit machine every pointer is 8 bytes and an `int` is 4 bytes. The compiler leaves a 4-byte gap after `data`, because `right` must start at an address that is a multiple of 8. For this reason `sizeof(struct node)` is 24 bytes; the value was verified by measuring on x86-64 with zig cc. So each node carries 20 bytes of overhead for 4 bytes of data. This is the price of the flexibility of linked structures.

The slides give the procedure `get_node()`, which allocates a node, only by its description: it allocates space with `malloc`, checks for `NULL`, and sets the children to `NULL`. An implementation that matches this description follows. All the driver programs for these notes used this procedure.

```
tree_ptr get_node(void) {
  tree_ptr r = malloc(sizeof *r);   /* 24 bytes */
  if (!r) {
    fprintf(stderr, "Out of memory\n");
    exit(EXIT_FAILURE);
  }
  r->left = r->right = NULL;
  return r;
}
```

Writing `malloc(sizeof *r)` takes the allocated size from the object the pointer points to rather than from the name of the type. Even if the name of the structure changes, the line stays correct. If `malloc` cannot find memory, it returns `NULL`. Writing to `r->left` in that case would be undefined behavior; the procedure therefore reports the error and ends the program. The two assignments in the last line are very important. `malloc` does not clear memory; the child pointers may hold a random address. Tree procedures recognize the end of a branch by `NULL`. A leaf whose children have not been cleared sends the procedures to a random place in memory.

Table 11.1 shows the memory view of a small tree with root 47, with sample addresses. The nodes of a tree do not have to sit next to each other in memory; each is in a separate place found by `malloc`. The shape of the tree is determined only by the addresses in the pointer fields.

Address	`left` (8 bytes)	`data` (4 + 4 padding)	`right` (8 bytes)
`0x1A40` (root)	`0x1B10`	47	`0x1A70`
`0x1A70`	`NULL`	55	`NULL`
`0x1B10`	`NULL`	26	`NULL`

Table 11.1: Memory view of a three-node tree with root 47, 26 on its left and 55 on its right. The addresses are only illustrative; each node is 24 bytes.

In the table the value of the root pointer `L` is `0x1A40`. `L->data` gives 47, and `L->left->data` gives 26. 26 sits after 55 in memory, but in the tree it is to the left of 55. There is no relation between the memory order of the nodes and their place in the tree.

11.2 Insertion

Insertion into a binary search tree is more restricted than insertion into a linked list. In a list a new element can go at the front, at the end, or in the middle. In a binary search tree the place of a new value is unique and is determined by the contents of the tree. This section first explains how this place is found, then how insertion order changes the shape of the tree. Then the insertion procedure is written in C and traced step by step.

11.2.1 The tree decides the place

In the slides the instructor makes this point with a classroom anecdote. A student is asked to name a value. The student also points out a place: "insert it to the right of 15." The instructor says that the place is the tree's business, not the student's. Indeed the signature of the insertion procedure takes only the value to insert and the tree; it takes no position information. The procedure searches for the new value starting at the root and inserts it where the search fails. If the value is not in the tree, the search ends at a `NULL` pointer. That `NULL` is the only place where the new value can sit without breaking the definition.

Although the place is unique, the same set of values can be represented by different trees. The shape of the tree depends on the order in which the values are inserted. The question in the slides asks about this with a small example.

Example 11.1

Question. Draw binary search trees of depth 3, 4, and 5 with the values 3, 5, 7, 9, 11. Can a tree of depth 2 be drawn?

Solution. The tree of depth 5 is the easiest: insert the values in increasing order. Each new value is larger than all the previous ones and is placed to the right of the rightmost node. The result is a chain extending to the right: 3 → 5 → 7 → 9 → 11. For depth 4 the order 5, 3, 7, 9, 11 can be used: 5 is the root, 3 goes to its left and 7 to its right; 9 and 11 again extend to the right. For depth 3 the order 9, 5, 3, 7, 11 is enough. The root is 9; 5 and 11 are its children; 3 and 7 are the children of 5. In preorder this tree is 9 5 3 7 11, and in parenthesized notation it is 9(5(3, 7), 11). The inorder traversal of all three trees is the same: 3 5 7 9 11.

A tree of depth 2 cannot be drawn. A binary tree of depth 2 has a root and at most two children of it; at most 3 nodes fit in total. Five values do not fit in this tree.

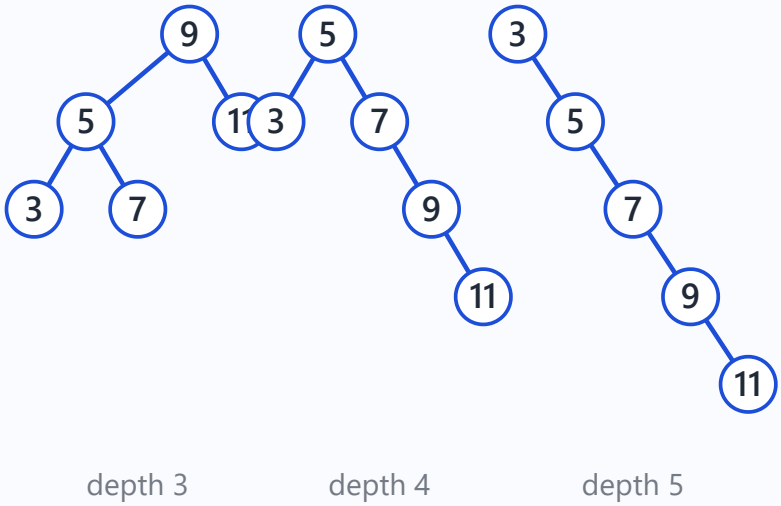


Figure 11.2: Three binary search trees of depth 3, 4, and 5 built with the values 3, 5, 7, 9, 11. The inorder traversal of all three is 3 5 7 9 11; only the insertion order differs.

The count in the last step of the example gives a general bound.

Theorem 11.2 (Depth and number of nodes)

A binary tree of depth k has at most $2^k - 1$ nodes. Accordingly, a binary tree with n nodes has depth at least $\lceil \lg(n + 1) \rceil$ and at most n.

Proof. The root is at level 1. Since every node has at most two children, level i has at most 2^{i-1} nodes. In a tree of depth k the levels run from 1 to k. The total is at most $1 + 2 + 4 + \ldots + 2^{k-1} = 2^k - 1$. For n nodes to fit we need $2^k - 1 \geq n$, that is, $k \geq \lg(n + 1)$; since k is an integer, $k \geq \lceil \lg(n + 1) \rceil$. For the upper bound it is enough that every level has at least one node: k levels need at least k nodes, so $k \leq n$. ■

For $n = 5$ the lower bound is $\lceil \lg 6 \rceil = 3$ and the upper bound is 5. Figure 11.2 shows these two ends and the depth in between. The meaning of the bounds is clear. In the worst case, a search in a tree looks at as many nodes as the depth. A tree whose depth is close to $\lceil \lg(n + 1) \rceil$ is as fast as binary search. A tree whose depth is close to n is no different from a linked list.

11.2.2 Insertion order changes the shape of the tree

The slides insert the same nine values into two empty trees in two different orders. The values are 17, 26, 34, 47, 49, 52, 55, 84, 93. The insertion order for the left tree is 47, 55, 49, 26, 34, 17, 52, 84, 93. The right tree gets the order 26, 17, 47, 93, 84, 52, 34, 55, 49. At every step the rule is the same: start at the root, go left if the new value is smaller than the one in the node and right if it is larger, and place it at the first `NULL`.

In the left tree the first value, 47, becomes the root. 55 is larger and goes to its right. 49 is larger than 47, so it goes right; there it is smaller than 55, so it is placed to the left of 55. 26 and 34 settle to the left of the root; 17 takes the left of 26. 52 follows the path 47 → 55 → 49 and goes to the right of 49; 84 goes to the right of 55, and 93 to the right of 84. The longest path in the end is 47 → 55 → 49 → 52, and the depth of the tree is 4.

In the right tree the root is 26. 17 goes to its left and 47 to its right. The next, 93, goes to the right of 47; 84 to the left of 93; and 52 to the left of 84. Since each new value is added below the previous one, the tree grows downward. 34 settles to the left of 47. 55 passes along the path 26 → 47 → 93 → 84 → 52 and is placed to the right of 52, and 49 to the left of 52. The longest path is 26 → 47 → 93 → 84 → 52 → 55 and the depth is 6. Figure 11.3 shows the state after the first four insertions and the final state.

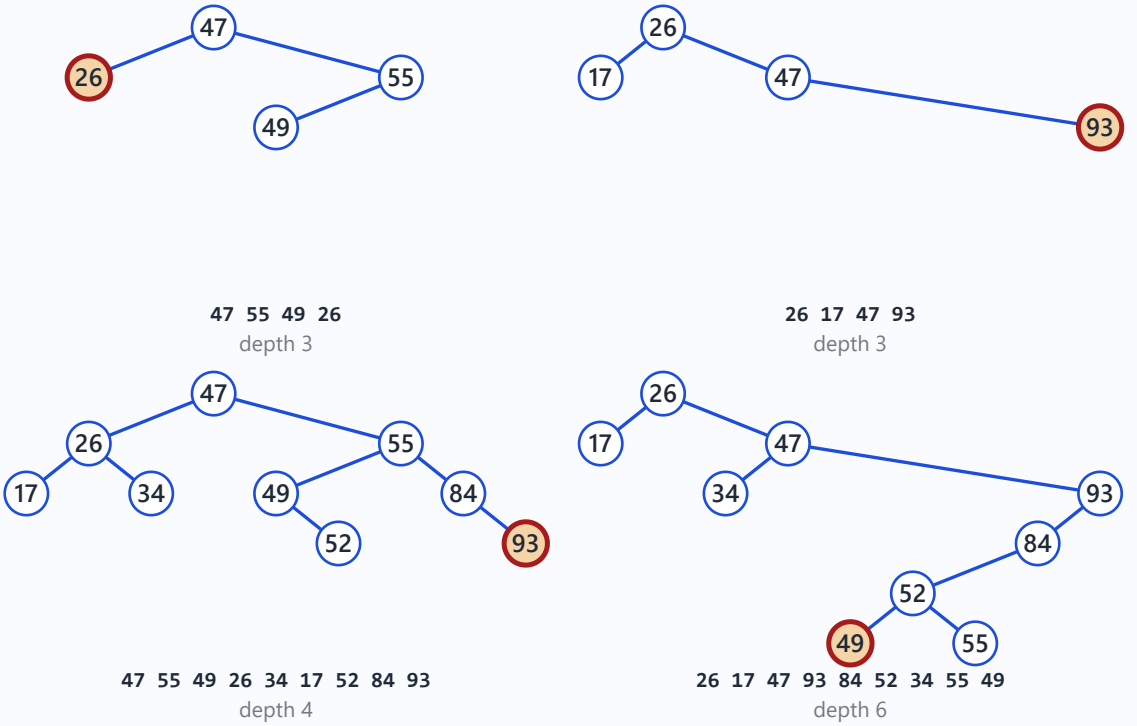


Figure 11.3: The same nine values are inserted into two empty trees in two different orders. Top: the first four values of each sequence have been inserted. Bottom: all values have been inserted; the depth of the left tree is 4 and that of the right tree is 6. The inorder traversal of both trees is the same: 17 26 34 47 49 52 55 84 93.

The two trees hold the same set and both satisfy the definition. By Theorem 11.1, their inorder traversals give the same sorted sequence. But their search costs differ. In the left tree the worst search looks at 4 nodes, in the right at 6. The insertion order determines the shape of the tree, and the shape determines the running time of all operations. In the words of the slides, the order in which data are inserted matters.

11.2.3 Skew and balanced trees

The worst insertion order is sorted input. When 1, 2, 3, ..., *n* are inserted in this order, every value is larger than all the previous ones, and each goes to the right of the rightmost node. The tree turns into a chain extending to the right. Data arriving in reverse order also builds a chain extending to the left. Trees whose nodes all extend in one direction are called **skewed trees**. The depth of a skewed tree is *n*, and every operation takes as long as in a linked list. Sorted data is more common in real life than one might think. Records arriving by date, student numbers given in increasing order, or a pre-sorted file are examples.

A tree whose depth stays around *lg n* is called a **balanced tree**. In a balanced binary search tree, insertion, deletion, and search take *O(lg n)*. The problem is that balance is not maintained automatically. Two kinds of solutions have been developed for this problem.

- **Self-balancing trees** rebalance the tree with local changes after every insertion and deletion. The AVL tree and the red-black tree are of this kind. The depth always stays *O(lg n)*.
- **Trees that move frequently used nodes toward the root** aim not at balance but at access speed. Each time a node is accessed, it is moved to the root by rotations. Frequently searched values thus gather near the root. This structure, called "zig-zag" in the slides, is known in the literature as the **splay tree**. It was proposed by D. Sleator and R. Tarjan in 1985. A single operation can be expensive, but over a long sequence of operations the average cost per operation is *O(lg n)*.

The instructor says this on the subject: if it is very important for the tree to be balanced, algorithms that build the tree in a balanced way should be used. These algorithms are not very hard, only a little longer. The last major section of these notes treats the best known of them, the AVL tree, in detail.

11.2.4 From tree to insertion order, from traversal to tree

At this point the slides ask two thought-provoking questions, marked with three question marks. The first is this: can the insertion order of the data be recovered from the picture of the tree? Is the tree formed from the incoming data unique?

The answer to the second half of the question is yes. The insertion procedure contains no randomness; when the same values arrive in the same order, the same tree is built every time. But the reverse direction is not unique. The order 2, 1, 3 builds the tree with root 2, left child 1, and right child 3. The order 2, 3, 1 builds the same tree, because 1 and 3 are not in each other's way and it does not matter which comes first. For an insertion order to build a given tree there is only one condition: every node must be inserted before its descendants. If an ancestor comes after a node among its descendants, that node settles somewhere else. Preorder always satisfies this condition, because it writes a node before all the nodes in its subtrees. So the preorder traversal of a tree is always a valid insertion order.

The second question is this: can the picture of the tree be recovered from the preorder, inorder, and postorder traversal sequences? In a general binary tree a single traversal sequence is not enough. For example, there are two trees with preorder A B: B can be the left child of A or the right child. If the

inorder sequence is given together with the preorder (or postorder) sequence, the tree is determined uniquely. The first element of the preorder sequence is the root. In the inorder sequence, the elements to the left of the root form the left subtree and those to the right form the right subtree. The same step is applied recursively to the subtrees.

In a binary search tree the job is easier. The inorder sequence is already known: it is the increasing order of the values. So the preorder sequence alone (or the postorder sequence alone) is enough to determine the tree. Even more briefly, the values in preorder are inserted in that order into an empty tree, and the tree is rebuilt.

Example 11.2

Question. The preorder traversal of a binary search tree is 30 20 10 25 40 35 50. What is the tree? What is its postorder traversal?

Solution. The first value, 30, is the root. Of the remaining values, those smaller than 30 (20, 10, 25) form the left subtree and those larger (40, 35, 50) form the right subtree. They also come in this order in the preorder sequence. The root of the left subtree is 20; 10 goes to its left and 25 to its right. The root of the right subtree is 40; 35 goes to its left and 50 to its right. The tree is 30(20(10, 25), 40(35, 50)) and its depth is 3. The postorder traversal writes the left subtree first (10 25 20), then the right subtree (35 50 40), and the root last: 10 25 20 35 50 40 30. Inserting the values in the same order into an empty tree gives the same tree; the result was verified with a driver program.

In the slides the instructor adds an observation on this subject. Before, binary tree procedures could not be tried by running code, because there was no real tree in the background. Once the insertion procedure is written, this obstacle disappears. Sample trees are built in a few lines; the traversal, copy, and equality procedures are really run on these trees. All the driver programs for these notes also built their trees this way.

11.2.5 The insertion procedure

The insertion algorithm rests on four observations. First, nothing can be inserted at a node that has both children; the new element goes into the empty place of a node that is missing at least one child. Second, the incoming value is searched for starting at the root. If the value in the node is larger than the new value, we move left; if smaller, right. Third, if the same value is found, no insertion is made, because the definition would be broken. Fourth, the new element is put in place of the first `NULL` that the search pointer meets. But when we reach `NULL`, the search pointer no longer points to any node. To remember the parent to which the new node will be attached, a second pointer trailing one step behind is needed. In the code this pointer is called `q`.

```
boolean bst_insert(int x, tree_ptr *L) {
  tree_ptr q = NULL, p = *L, r; // q trails p
  while (p) {                   // down to NULL
    q = p;                      // q follows p
    if (x < p->data) p = p->left;
    else if (x > p->data) p = p->right;
    else { printf("Duplicate key.\n");
           return FALSE; }
  }
  r = get_node();                  // children NULL
  r->data = x;
  if (*L == NULL) *L = r;          // empty: root
  else if (x < q->data) q->left = r;
  else q->right = r;
  return TRUE;
}
```

The lines of the code work as follows.

- **Line 2.** Three local pointers are defined. `p` starts from the root of the tree; `*L` is the value of the caller's root pointer. `q` stays one step behind `p` and is `NULL` at first. `r` will hold the new node.
- **Lines 3–9.** The loop continues until `p` is `NULL`. On each iteration the trailing pointer is first updated with `q = p`, and then `p` descends one level. If `x` is smaller than the value in the node, we move to the left child; if larger, to the right child. If it is neither, the values are equal: the procedure prints a warning and returns `FALSE`. In this case nothing in the tree changes and no memory is allocated.
- **Lines 10–11.** When the loop ends, `p` is `NULL` and `q` points to the node that will become the parent of the new node. The new node is allocated with `get_node()`; its children are `NULL`. `x` is written into the data field.
- **Line 12.** If the tree is empty, the loop never ran and `q` is still `NULL`. The new node becomes the root: `*L = r`. This assignment changes the caller's root variable.
- **Lines 13–14.** If the tree is not empty, `x` is compared once more with the parent's value, and the new node is attached to the parent's left or right pointer.

In the loop it is known which direction `p` went, but this information is lost when the loop ends. The second comparison in line 13 reproduces it. It would also be possible to keep a direction variable in the loop. The code chooses brevity. Only one extra comparison is made and the cost does not change.

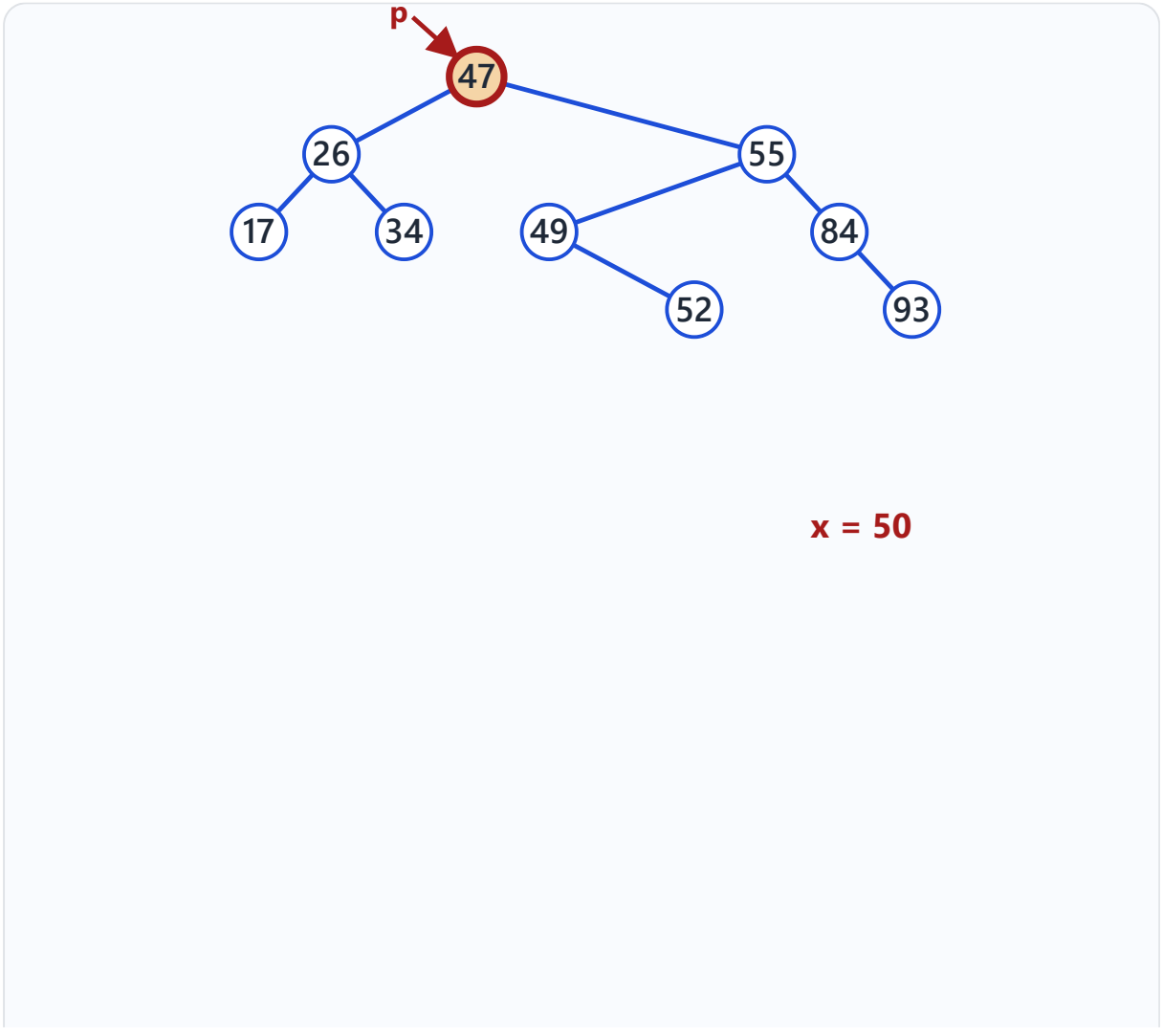
11.2.6 Code trace: inserting 50

The slides trace the insertion of 50 into the left tree (47, 55, 49, 26, 34, 17, 52, 84, 93). Table 11.2 gives the state of the variables at each step.

Step	Line	`q`	`p`	Explanation
0	2	NULL	47	`p` is at the root; `q` does not point to any node yet.
1	4–6	47	55	*50 > 47*: its place is in the right subtree.
2	4–5	55	49	*50 < 55*: descend into the left subtree.
3	4–6	49	52	*50 > 49*: descend to the right.
4	4–5	52	NULL	*50 < 52*: the left of 52 is empty; the loop ends.
5	10–11	52	NULL	The new node `r` is allocated; `r->data = 50`.
6	13	52	NULL	The tree is not empty and *50 < 52*: `q->left = r`.
7	15	—	—	`TRUE` is returned.

Table 11.2: Trace of the call `bst_insert(50, &L)`. The values name the nodes; "47" is written instead of "node(47)".

The four comparisons in the table correspond to the four nodes on the path from the root to 52. Before the insertion the depth of the tree is 4; 50 enters at the fifth level, below the path 47 → 55 → 49 → 52, and the depth becomes 5. The insertion changed a single pointer: the left child of 52. The rest of the tree was not touched. Figure 11.4 shows four frames of the trace.



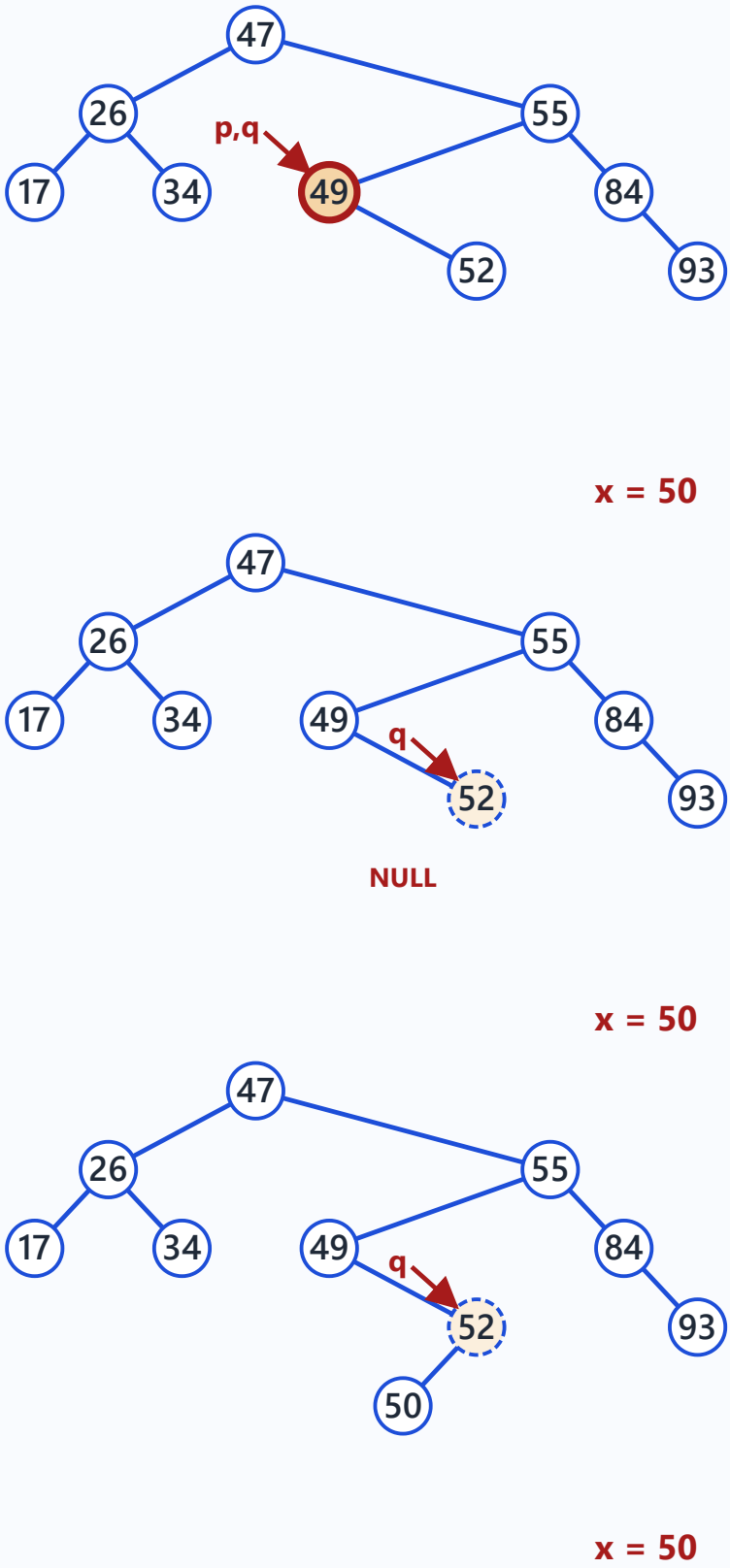


Figure 11.4: Four frames from the trace of `bst_insert(50, &L)`. Left to right and top to bottom: `p` at the root and `q` `NULL`; `p` has descended to 52 and `q` is at 49; `p` has become `NULL` and `q` points to 52; the new node `r` has been attached to the left of 52.

If the same call had been made with 47, equality would have been found on the first iteration of the loop. The procedure would print "Duplicate key." and return `FALSE`. These two cases were tried in the driver program: the return value is 1 for 50 and 0 for 47. After the insertion, the preorder traversal of the tree is 47 26 17 34 55 49 52 50 84 93.

11.2.7 Why `tree_ptr *L`?

The insertion procedure takes the tree not as `tree_ptr L` but as `tree_ptr *L`. That is, it asks for the address of the root pointer, not its value. The reason is the empty tree. If the tree is empty (`*L == NULL`), the inserted element becomes the root, and the caller's root variable must point to this new node. In C all parameters are passed by value. If the procedure took `tree_ptr L`, the assignment `L = r` would change only the procedure's local copy. The caller's `root` variable would stay `NULL`, and the allocated node would leak, since nothing points to it.

To let a procedure change a variable, the address of the variable is passed. This technique is called **call by reference**. Since the root variable is already a pointer, its address is a **pointer to a pointer**. The call is written `bst_insert(x, &root)`. Inside the procedure, `*L` is the caller's `root` variable itself; `*L = r` writes to that variable.

There is a second solution to the same problem: the procedure gives the new root of the tree as its return value. The slides show this recursive version as well.

```
/* Recursive version: returns the new root */
tree_ptr insert(tree_ptr L, int x) {
  if (!L) {
    L = get_node();
    L->data = x;
  } else if (x < L->data)
    L->left = insert(L->left, x);
  else if (x > L->data)
    L->right = insert(L->right, x);
  return L;        /* root = insert(root, x); */
}
```

When the procedure reaches an empty subtree, it allocates the new node and returns it. At a non-empty node, it inserts into the left or right subtree according to the value. It assigns the result of the recursive call back to the corresponding child pointer. If the subtree is not empty, the recursive call returns the same pointer and the assignment changes nothing. If the subtree is empty, the recursive call returns the new node and the assignment attaches it to the tree. If the values are equal, both branches are skipped and the node is returned as it is. The usage is `root = insert(root, x);`. If the caller forgets to assign the return value, the first insertion into an empty tree is lost. This version has no error report either: an equal value is silently rejected. In the driver program, nine values were inserted with each of the two procedures separately; the preorder traversals of the two trees gave the same output.

The cost of the insertion procedure is the length of the path the new value descends. At each level one node is examined and a constant amount of work is done. If the depth of the tree is h, insertion takes $O(h)$. In a balanced tree $h = O(\lg n)$, and in a skewed tree $h = n$. The slides write this range as $\Omega(\lg n) – O(n)$. The lower bound comes from the fact that no tree can have depth smaller than $\lceil \lg(n + 1) \rceil$. For example, in a tree with all levels full, all the null pointers are below the last level; every insertion examines $\lg(n + 1)$ nodes. The iterative version uses only three pointers of extra memory, that is, $O(1)$. The recursive version opens a call frame for each level and needs $O(h)$ extra memory. On a skewed tree built from 2000 sorted values the iterative procedure ran without trouble. The recursive procedure makes 2000 nested calls at this depth. The default 1 MB call stack of Windows even held a chain of 30,000 nodes. But the call depth grows with n; for a long enough chain the stack overflows.

The slides have one more detail, called the "C99 fix". In the original code in the book, `q` is not given an initial value. When the tree is empty the loop never runs and `q` stays with an undefined value. The code avoids using this value through the check `*L == NULL`. Even so, compilers give a "may be used uninitialized" warning. The code in the slides starts with `q = NULL`; this removes the warning and makes the code safe.

11.3 Search

Search is the reason the binary search tree exists. Insertion and deletion also start with a search: insertion looks for the place of the new value, deletion looks for the node to be deleted. This section writes search first with a loop and then with recursion. It then treats a subtle point in recursive search and search in a general, unsorted binary tree.

11.3.1 Iterative search

The sought value `x` is first compared with the value at the root. If they are equal, the search ends. If `x` is smaller, the sought value, if present, is in the left subtree; by definition every value in the right subtree is larger than the root and therefore larger than `x`. If `x` is larger, the same reasoning points to the right subtree. The same step is repeated in the chosen subtree. If a `NULL` pointer is reached, the value is not in the tree.

```
tree_ptr bst_search(int x, tree_ptr L) {
  while (L) {           // pick a subtree
    if (x == L->data) return L;
    if (x < L->data) L = L->left;
    else             L = L->right;
  }
  return NULL;          // not found
}
```

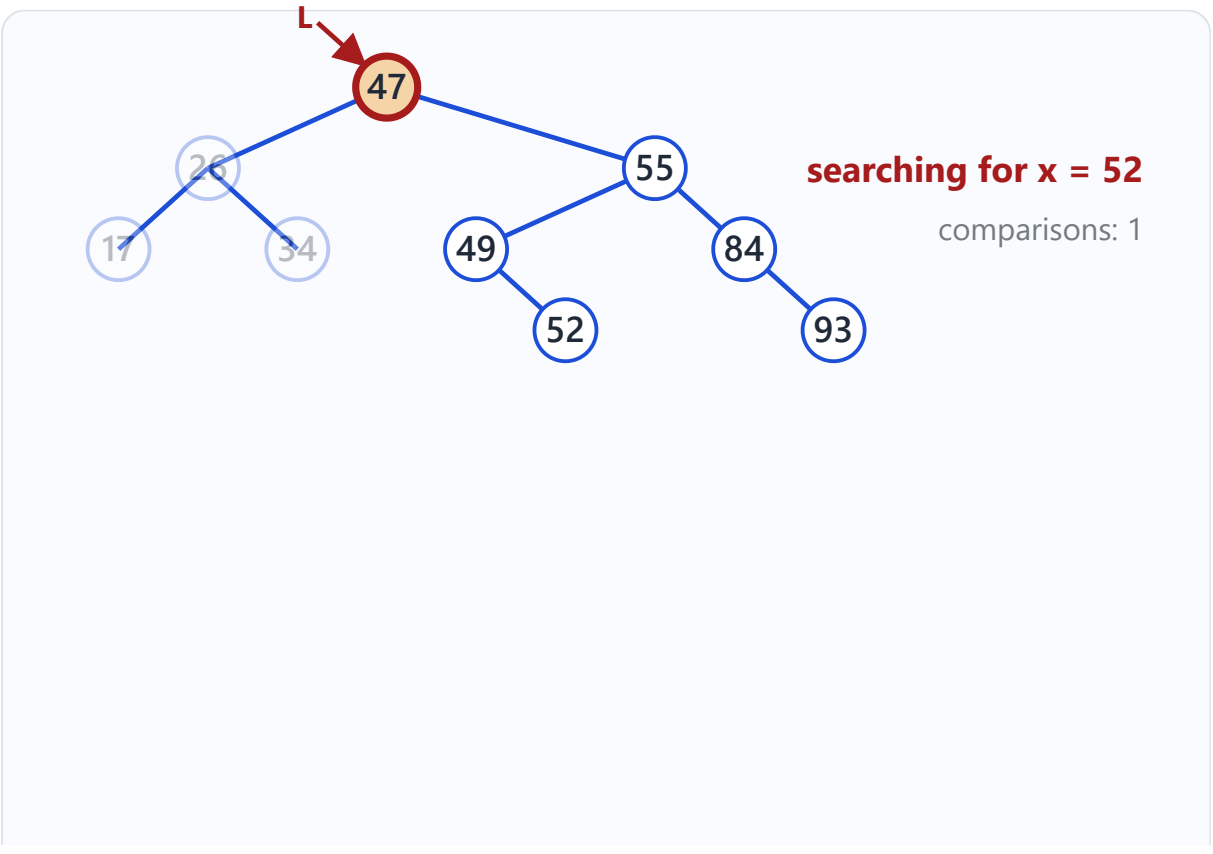
The procedure uses the pointer `L`, which it receives by value, as a traveler through the tree. The loop in line 2 continues as long as `L` is not `NULL`. Line 3 checks equality and returns the address of the node found. Lines 4 and 5 move `L` down to the appropriate child. If the loop is left (line 7), the value is absent and `NULL` is returned. Changing `L` here does not affect the caller; the caller's root pointer stays where it is. Unlike insertion, search does not need the address to be passed, because search does not change the tree.

The address of the node is chosen as the return value. This way the caller learns both whether the value exists and gets access to the other fields in the node. When a record is searched for by key, it is usually these other fields that are really wanted.

Example 11.3

Question. In the tree before 50 is inserted (47, 55, 49, 26, 34, 17, 52, 84, 93), which nodes does `bst_search(52, L)` examine?

Solution. At the root *52 > 47* and `L` goes right, to 55. The entire left subtree (26, 17, 34) is eliminated. At 55, *52 < 55* and `L` goes left, to 49. 84 and 93 are never examined. At 49, *52 > 49* and `L` goes right, to 52. At 52 equality is found and the address of the node is returned. Four nodes were examined. In the C code, equality is checked first at each node, then less-than. At the first three nodes two checks are made, at the last one check; in all, that is seven comparison operations. In the analysis each node visit is counted as a single three-way comparison: less, equal, or greater.



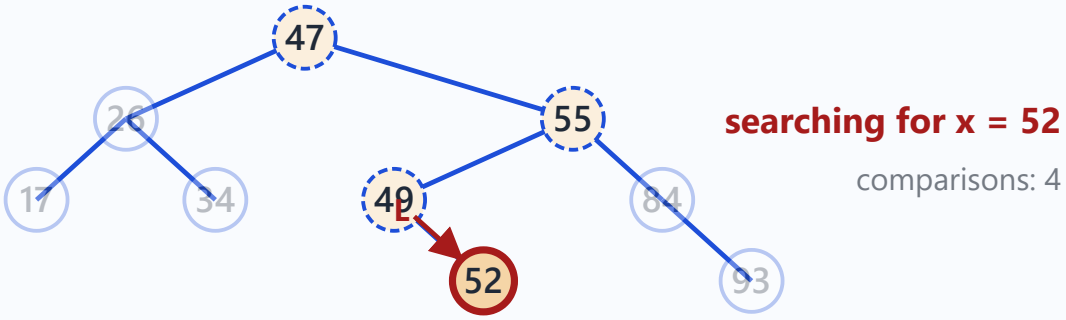


Figure 11.5: The first and last frames of the trace of `bst_search(52, L)`. Top: at the root the comparison *52 > 47* eliminates the left subtree (faded nodes). Bottom: the path has advanced along 47 → 55 → 49 → 52 and the value has been found with four comparisons; the right subtree of 55 was never examined.

Figure 11.5 shows the power of search visually: each comparison eliminates an entire subtree. If the tree is balanced, each elimination discards about half of the remaining nodes. This is the behavior of binary search in a sorted array. When a value that is not in the tree is searched for, the same path is followed. For example, if 60 is searched for, the path 47 → 55 → 84 is examined. Since the left child of 84 is `NULL`, the search returns `NULL` after three nodes.

The cost of search is the length of the path examined and is bounded by the depth h of the tree: $O(h)$. In a balanced tree this is $O(\lg n)$, and in a skewed tree $O(n)$. The iterative procedure uses only `L` as extra memory: $O(1)$.

Next to this procedure in the slides there is an analogy by the instructor. When a manager defines a task for a clerk, he must speak very clearly; the clerk should not have to knock on the door afterwards to ask questions. The definition of a procedure is like that. The contract of `bst_search` is clear: it returns the address of the node if the value exists and `NULL` if not. It works correctly on the empty tree, does not change the tree, and does not touch a global variable. Whoever uses the procedure can use it correctly without looking inside.

11.3.2 Recursive search

Since the definition is recursive, search is naturally written recursively too. In the empty tree the value is absent. If the value is found at the root, the job is done. Otherwise the same question is asked of the appropriate subtree and its answer is passed up unchanged.

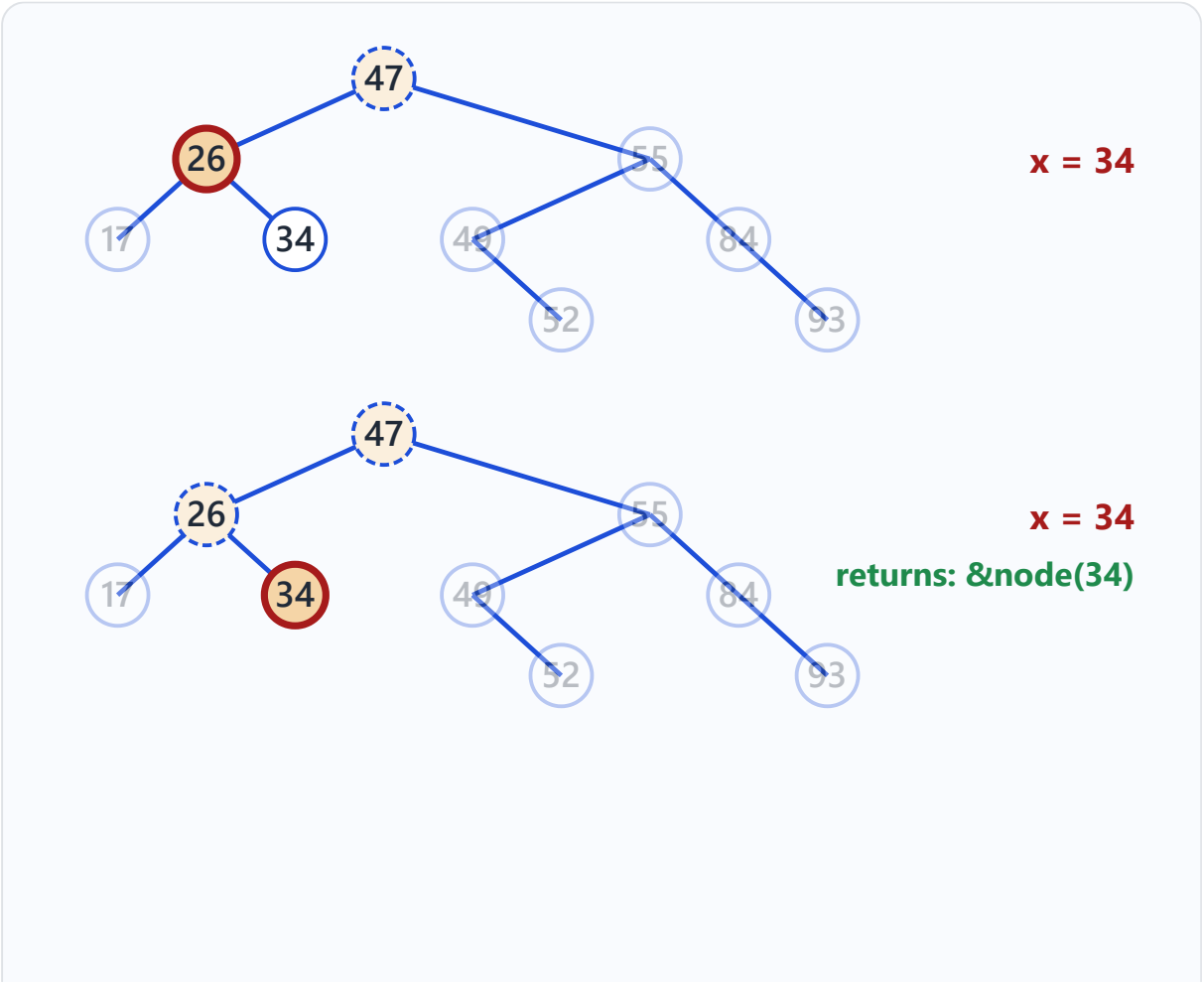
```
tree_ptr search(int x, tree_ptr L) {
  if (!L) return NULL;          // not found
  if (x == L->data) return L;   // found
  if (x < L->data)
    return search(x, L->left);
  return search(x, L->right);
}
```

The first two lines are the base cases. If the tree is empty (line 2), `NULL` is returned; if the value is at the root (line 3), the address of the root is returned. Lines 4–6 are the recursive step. Both calls are preceded by `return`. Whatever the recursive call returns, this frame returns as well, and it does no other work. When the last thing a procedure does is call itself, the call is called **tail recursion**. Such a procedure is easily turned into a loop: the parameter is updated with its new value and control returns to the top. Iterative search is exactly this procedure obtained by that transformation. Compilers that optimize well can make this transformation themselves.

Frame	`L`	Comparison	What it does	Returns
1	47	*34 < 47*	`return search(x, L->left)`	&node(34)
2	26	*34 > 26*	`return search(x, L->right)`	&node(34)
3	34	*34 = 34*	`return L`	&node(34)

Table 11.3: Frame trace of the call `search(34, L)`. Each row is a call frame; the last column is the value the frame returns.

Table 11.3 summarizes the descent and the return of the calls. Three frames go down, the bottom frame finds the node and returns its address. On the way back the frames do no computation; they pass the address they receive unchanged to the frame above. Figure 11.6 shows three moments: the middle of the descent, the moment of finding, and the moment the result reaches the root.



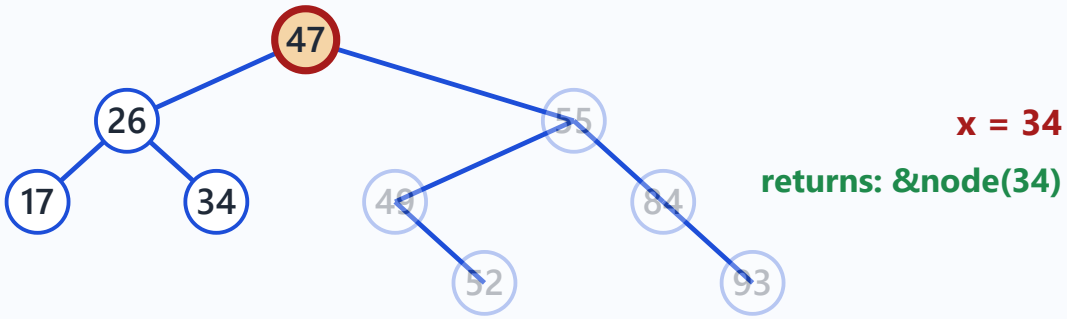


Figure 11.6: Recursive `search(34, L)`. Top: the frames for 47 and 26 wait on the stack; at 26 *34 > 26*, so the call goes right. Middle: 34 is found. Bottom: the address has been carried up unchanged through two frames and has reached the root.

The time cost is the same as the iterative version: *O(h)*. The memory cost is different. Each call opens a frame, and at the deepest point *h* frames sit on the stack at the same time. The extra memory is *O(h)*. In a balanced tree this is a few dozen frames and is negligible. In a skewed tree the call depth reaches *n*.

11.3.3 A trap in recursive search

The slides show the body below, written by a student, and ask "Does this example work?"

```
if (L) {
  if (x == L->data) return L;
  if (x <  L->data) return search(x, L->left);
  return search(x, L->right);
}
return NULL;   /* L == NULL: empty (sub)tree */
```

The real question the code asks is in line 3. If `x` is smaller than the root, the `if` block in line 3 is entered and the search is made in the left subtree. Whatever the recursive call returns, the procedure returns. The search of the right subtree in line 4 is **never** done in this case. If the value is not found in the left subtree, `NULL` is returned and the procedure ends without looking right at all.

In a binary search tree this behavior is exactly what is wanted. If `x` is smaller than the root it cannot be in the right subtree; looking there is wasted work. The code is correct and *O(h)*. In a general, unsorted binary tree the situation is different. `x` may be in the right subtree even though it is smaller than the root; this code misses it. In the driver program, 2 was searched for in a tree whose root is 5, left child 9, and right child 2. Because *2 < 5*, the search went left, reached `NULL` at the left of 9, and answered "not found".

The `return NULL` in the last line also needs attention. In the original version of the slides there was a comment next to this line: "It never gets here, but whatever, let's write it anyway...". The comment was wrong and was corrected in the slides. When `L` is `NULL`, the `if` block is skipped and the

procedure arrives exactly at this line. This happens in the last call of every unsuccessful search. If the line is deleted, the procedure ends on the empty tree without returning a value. Reaching the end of a procedure that must return a value and then using that value is undefined behavior in C. The compiler gives a warning in this case; the caller may get a random address.

The instructor has one more note on this example: when writing recursive code, sometimes work is also done after the `return` line, as in the example. This recalls two separate forms of recursion. In search, the result of the recursive call is passed on unchanged and no work remains after the call. When the depth or the number of nodes is computed, the results of the recursive calls are combined and the computation is done on the way back. Which form is needed depends on whether the problem needs both subtrees.

Common mistake

Not writing `return` in front of the recursive call in recursive search: `if (x < L->data) search(x, L->left);`. The recursive call finds the node and returns its address, but the calling frame discards this value without using it. The procedure then moves on to the next line and returns a wrong value. The code usually compiles; the compiler gives only a "not every path returns a value" warning. Warnings should be turned on and taken seriously.

11.3.4 Searching any binary tree

In a binary tree whose values are not sorted, search cannot know which direction to go. Both subtrees must be examined. The sample solution in the slides first looks at the left subtree; if it does not find the value there, it moves on to the right.

```
tree_ptr search_rec(int data, tree_ptr L) {
  if (!L) return NULL;              /* empty tree?   */
  if (L->data == data) return L;    /* key found     */
  tree_ptr T = search_rec(data, L->left);
  return T ? T : search_rec(data, L->right);
}
```

The first two lines of the procedure are the same as in recursive search. Line 4 searches the left subtree and stores the result in `T`. The conditional expression in line 5 returns `T` if it is not `NULL`. If the value was found on the left, the right is not looked at. If it was not found on the left, the result of the search of the right subtree is returned. If it is not on the right either, this result is `NULL`.

The procedure does not use the ordering information of the tree. If the value is not in the tree or is at the last node visited, all nodes are examined; the worst case is *O(n)*. This procedure also works correctly on a binary search tree, but it is needlessly slow. In the driver program, 2, 1, and 7 were

searched for in a general tree with root 5, left child 9, right child 1 of 9, and right child 2 of the root. The procedure found 2 and 1 and returned `NULL` for 7. In the same tree, the binary search tree procedures could not find 2.

The memory cost of this procedure is the call depth: *O(h)*. The procedure is also used in the solution of the exam question on finding a weight. The instructor says something ambitious about these procedures: everyone in the class should be able to write them with eyes closed. Then he softens it: "All right, at least with one eye open." What is meant is this: the search, insertion, and depth procedures are the alphabet of trees. Every bigger problem is solved by combining them.

11.4 Depth

The depth of a tree determines the cost of all operations. The procedure that computes it is small but an instructive example. The problem needs the answers from both subtrees and gives one of the cleanest examples of recursive thinking. This section first treats an iterative solution, then the recursive solution and two wrong ways of using it.

11.4.1 Iterative depth and a global variable

The slides open this topic with a question: write the depth iteratively. The instructor reacts sharply to the typical solutions seen in class. Those solutions traverse the tree with a stack and keep the largest level seen so far in a **global variable**. According to the instructor, this code earns no points even if it works. A student asks, "If it works correctly, won't you give even one point?" The answer is firm: not even a quarter of a point. Why he is so strict will be explained shortly. First let us see an iterative solution that uses no global variable.

```
/* iterative, no globals:
walk level by level, count levels */
int depth_iter(tree_ptr L) {
  tree_ptr k[MAX_SIZE];
  int front = 0, rear = 0, d = 0;
  if (L) k[rear++] = L;
  while (front < rear) {          /* one level */
    int last = rear;
    d++;
    for (; front < last; front++) {
      if (k[front]->left) k[rear++] = k[front]->left;
      if (k[front]->right) k[rear++] = k[front]->right;
    }
  }
  return d;       /* needs MAX_SIZE >= n */
}
```

The solution traverses the tree in **level order**. The array `k` is used like a queue: `front` is the index of the next element and `rear` is the index of the first empty cell. The root is put into the queue. Each iteration of the outer loop handles one level. At the start of the iteration the end of that level is marked with `last = rear` and `d` is increased by one. The inner loop takes the nodes of that level from the queue in turn and adds their children to the back of the queue. These children are the next level. When the queue is empty, all levels have been counted and `d` gives the depth. In the empty tree nothing is put into the queue and `d = 0` is returned.

An element taken from the queue is not deleted; `front` only advances. The array therefore has to be large enough to take all the nodes of the tree: `MAX_SIZE ≥ n`. The comment in the last line of the code recalls this condition. Since each node enters the queue once and leaves once, the time is *O(n)*. The memory is the size of the array and sits in the stack region. In the driver program `MAX_SIZE` was chosen as 60,000; with 8-byte pointers this is about 470 KB. Since the default call stack on Windows is 1 MB, an array of a few hundred thousand nodes cannot be defined locally. In a real program this array should be allocated with `malloc`. The procedure gave the same result as the recursive version on the sample trees in the slides and on a tree built from 50,000 random values.

The reason for the instructor's reaction to the global variable is the scale of software. In large companies big jobs are done together. A procedure written by one person is used by someone at the other end. The instructor explains this with an analogy: the person using the procedure does not know the person who wrote it; perhaps he is in another country. He cannot pick up the phone and ask, "You have a variable called `max`; what is going on?" A procedure that uses a global variable has three problems.

- **Hidden state.** The result of the procedure depends not only on its parameters but also on the current value of the global variable. If the variable holds a value left over from a previous call, the second call gives a wrong result. The user has to know that the variable must be reset before every call.
- **Name clashes.** If a global variable with the same name is defined in two separate files, the linker reports an error or, worse, silently merges the two variables.
- **Lack of reentrancy.** The procedure cannot be run for two trees at the same time. Two threads corrupt the same global variable; the procedure also cannot call itself recursively for another tree.

The slides draw three principles from this discussion. First, a procedure should be complete in itself: it takes its input from parameters, gives its output as a return value, and keeps no hidden state. Second, a **local prototype** of another procedure can be declared inside a procedure. This only makes the name visible inside that procedure; the definition stays outside. Standard C does not allow nested procedure definitions. This technique is used in the solution of the exam question on finding a weight. Third, in a recursive solution each call has its own local left and right variables; there is no need for a global counter at all.

11.4.2 Recursive depth

The recursive definition of depth is a single sentence. The depth of the empty tree is 0. The depth of a non-empty tree is 1 more than the depth of the deeper of its left and right subtrees: *depth(L) = MAX(depth(left), depth(right)) + 1*. The added 1 is the root itself. The code follows this definition exactly.

```
int depth(tree_ptr L) {
  int left, right;
  if (L) {                    // pointer NULL?
    left = depth(L->left);
    right = depth(L->right);
    return MAX(left, right) + 1;
  }
  else return 0;              // reached past a leaf
}
```

The two local variables in line 2 store the depths of the subtrees. Line 3 asks whether the problem has become small enough. The instructor stresses this question in particular: "Has the problem become small enough? `NULL` may come at the start too!" The `NULL` check is not only for the empty children below the leaves; in the first call the tree itself may be empty. Lines 4 and 5 compute the depths of the two subtrees. Line 6 takes the larger and adds 1. Line 8 returns 0 for the empty tree.

`MAX` in the slides is a macro: `#define MAX(a,b) ((a) > (b) ? (a) : (b))`. The arguments here are just local variables; it does no harm that the macro evaluates an argument twice. Why the same macro is dangerous in the one-line version is the subject of the next subsection.

The slides trace the procedure on a tree with six nodes. A is the root, with B on its left and C on its right. D is on the left of B, E on the left of C, and F on the left of E. Table 11.4 gives the `{left, right}` values of each frame and its return value.

Finishing order	`L`	`left`	`right`	Returns
1	D	0	0	1
2	B	1	0	2
3	F	0	0	1
4	E	1	0	2
5	C	2	0	3
6	A	2	3	4

Table 11.4: Trace of the call `depth(A)`. The rows are given in the order in which the frames finish; calls made with `NULL` return 0 and are not shown separately in the table.

The calls follow the left branch first. `depth(A)` calls `depth(B)`, which calls `depth(D)`. Both children of D are `NULL`; the two recursive calls return 0 and D returns 1. B has no right child; B returns *MAX(1, 0) + 1 = 2*. The `left` variable of A becomes 2, and now `depth(C)` is called for the right subtree. From C we descend to E, and from E to F. F returns 1, E returns 2, and C returns 3. Finally A returns *MAX(2, 3) + 1 = 4*. The driver program gave the result 4 on the same tree.

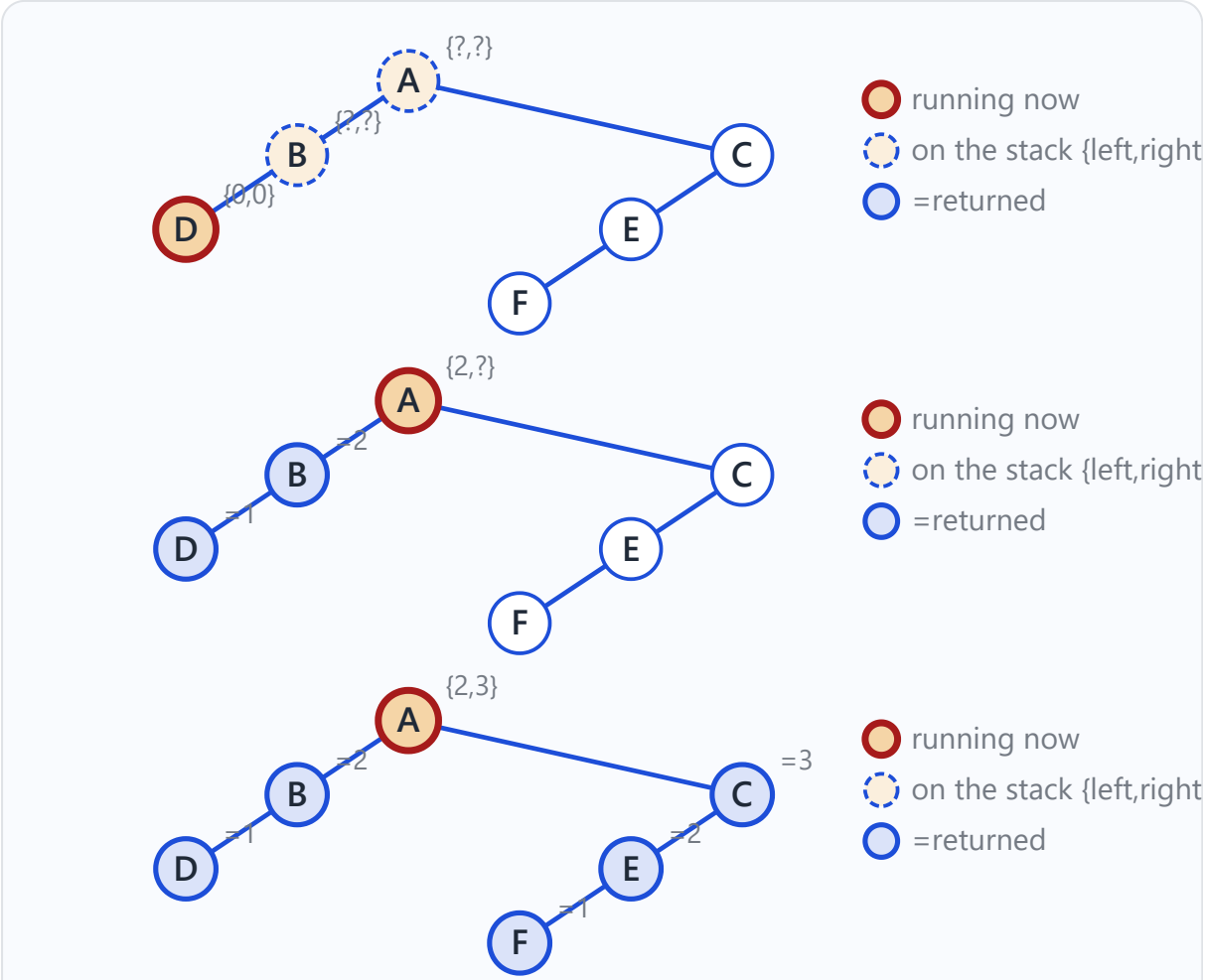


Figure 11.7: Three frames from the trace of recursive depth. Top: D is running; the frames of A and B wait on the stack with the values `{?, ?}`. Middle: B has returned 2, the `left` variable of A has become 2, and it is C's turn. Bottom: all frames have finished and A returns 4.

Figure 11.7 shows a property of the frames. Each node has its own two local variables: the left and right depths. When a frame finishes, its local variables vanish too. Calls made with `NULL` discard their local variables without ever using them. On the stack at any moment there are only the frames on the path from the root to the current node: at most *h + 1* frames.

The cost analysis is as follows. One call is made for each node. In addition, one call is made for each `NULL` child. A binary tree with *n* nodes has *n + 1* empty child pointers: of the *2n* pointers in total, *n − 1* point to a node and the rest are `NULL`. The number of calls is exactly *2n + 1*. In the driver program

this number was verified with a counter for every chain length from 1 to 24 and for perfect trees up to 262,143 nodes. Since each call does constant work, the time is $\Theta(n)$. To find the depth, looking at every node is necessary; no procedure can be faster than this. The memory is the call depth: $O(h)$.

11.4.3 The one-line version and the macro trap

The same procedure can also be written as a single expression without local variables. The slides present this version as a "professional" way of writing, and then show the trap inside it.

```
int depth(tree_ptr L) {    /* one line */
  return !L ? 0 : MAX(depth(L->left),
                      depth(L->right)) + 1;
}
```

The code is readable and gives the right result. But `MAX` is not a function but a **macro**. Before compilation the preprocessor replaces `MAX(a, b)` textually with `((a) > (b) ? (a) : (b))`. Here `a` and `b` are procedure calls. The expanded expression first runs both calls for the comparison. Then, to return the larger one, it runs that call *once more*. At every node one of the subtrees is computed twice.

In a skewed tree the price of this is exponential. In a chain extending to the right, the left child of every node is `NULL`. Since the depth of the right subtree is larger, the right call is made twice. If the number of calls in a chain of n nodes is $T(n)$, then $T(n) = 2T(n-1) + 2$. The 2 here is the node's own call plus the call for the `NULL` left child. The solution is $T(n) = 3 \cdot 2^n - 2$. The situation is the same in a chain extending to the left. There the right child is `NULL`, the left call turns out larger and is made twice. In a full, balanced tree the depths of the two subtrees are equal. The comparison comes out false and the right subtree is recomputed. This time $T(n) \approx 3T(n/2)$ and the solution is $\Theta(n^{\lg 3}) \approx \Theta(n^{1.585})$. So even in a balanced tree the time is not linear.

Tree	n	With local variables	One-line (macro)
chain	4	9	46
chain	12	25	12.286
chain	20	41	3.145.726
chain	24	49	50.331.646
full	1.023	2.047	88.573
full	65.535	131.071	64.570.081

Table 11.5: Call counts of the two depth procedures (measured with a counter). The version with local variables makes $2n + 1$ calls on every tree.

Table 11.5 shows the difference in numbers. On a chain of 24 nodes the two procedures give the same result (24). One makes 49 calls, the other more than fifty million. Every new node doubles the number of calls of the macro version. On a chain of 40 nodes the number is $3 \cdot 2^{40} - 2$, which exceeds three trillion. All the measured values fit the formula $3 \cdot 2^{n} - 2$ exactly. On perfect trees the macro version makes $(3^{k+1} - 1)/2$ calls, where k is the number of levels.

The solution is to use a real function instead of a macro. A function evaluates its arguments once before the call and copies the values. The solution of the exam question on finding a weight takes this route:

```
int max(int a, int b) { return a > b ? a : b; }
```

The one-line depth procedure written with this function made exactly $2n + 1$ calls on every tree in the driver program. The second solution is the version with local variables. The values are first taken into variables; the macro now only compares variables.

> **Common mistake**
>
> Passing an expression with side effects or a cost to a macro as an argument: `MAX(depth(...), depth(...))`, `MAX(i++, j)`, `SWAP(a[i++], b, t)`. Because the macro copies its argument as text, the expression is executed more than once. With recursive calls the result is exponential time, and with increment operators it is a wrong value. Only simple variables should be given to a macro; otherwise a function should be used.

Next to this subsection in the slides there is one more observation by the instructor. According to him, it is hard to make engineers accept recursive thinking and solving recurrence relations. Someone used to thinking in loops says, "What nice code I'm writing," and resists a new way of thinking. The depth example shows why this resistance is misplaced. The iterative solution needs a queue, a size limit, and two nested loops. The recursive solution is the definition itself and fits in four lines.

11.4.4 Adding an extra parameter

The slides open this subsection with a well-known saying of A. J. Perlis. Roughly, it says: if you have a procedure with ten parameters, you have probably forgotten some. The joke says that a bloated parameter list is a sign of a design mistake.

A student asks: would it be all right to add an extra parameter, `int depth(tree_ptr L, int a)`, and call the procedure as `depth(L, 0)`? The solution the student has in mind is probably this: `a` carries the level descended so far, `a + 1` is passed to each recursive call, and when `NULL` is reached, `a` is returned. The code may work. But the instructor does not accept this design. Why is the extra parameter defined, and why is zero passed in the first call? What will we tell the user of the

procedure: "Don't you dare forget to pass zero to `a`!"? The instructor makes this concrete with an example: could there be a sorting procedure that sorts when 0 is passed and fails to sort when 1 is passed?

The problem is that the parameter is not an *input* but an initial value belonging to the internal workings of the procedure. It puts an obligation on the user, and when the obligation is forgotten the procedure silently gives a wrong result. If necessary, such an extra parameter is kept in a helper procedure that is not exposed. The procedure exposed to the user takes only `L` and itself calls the helper with the correct initial value.

Not every extra parameter is bad. A sorting procedure can be given a parameter telling whether to sort from small to large or from large to small (−1/+1 or `'<'` / `'>'`). A parameter giving the type of the data can also be passed. These parameters *select the behavior* of the procedure. Each of their values is meaningful, and the user makes the choice knowingly. The principle is this: a parameter should select behavior and should not carry a hidden initial value.

11.5 Deletion

Deletion is the most delicate operation of the binary search tree. Insertion is always done in place of a `NULL` pointer and changes only that pointer. The node to be deleted, however, can be anywhere in the tree, for example at the root. After the node is removed, the remaining nodes must be reconnected and the tree must again satisfy the definition.

11.5.1 Three cases

There are three cases, according to the number of children of the node to be deleted.

- **A node with no children (a leaf).** The parent's pointer to this node is set to `NULL` and the node is freed with `free`. The rest of the tree is not affected.
- **A node with one child.** The parent's pointer is changed to point to the only child of the deleted node. The subtree of the child moves up one level as a whole. This does not break the order. If the deleted node is to the left of its parent, every value in its subtree is smaller than the parent. When the subtree moves up, it is still to the left of the parent.
- **A node with two children.** The parent's single pointer cannot point to both subtrees at once. In place of the deleted node, a value that fits between the two subtrees must be put. There are different methods for this.

The slides examine the two-children case on a sample tree. The values 40, 20, 60, 10, 30, 50, 70, 42, 55, 65, 73, 52, 57, 66, 56 were inserted into the tree in this order. The depth of the tree is 6; the longest path is 40 → 60 → 50 → 55 → 57 → 56. The node to be deleted is 60. 60 has two children: 50 and 70. Under each of them there is a sizable subtree.

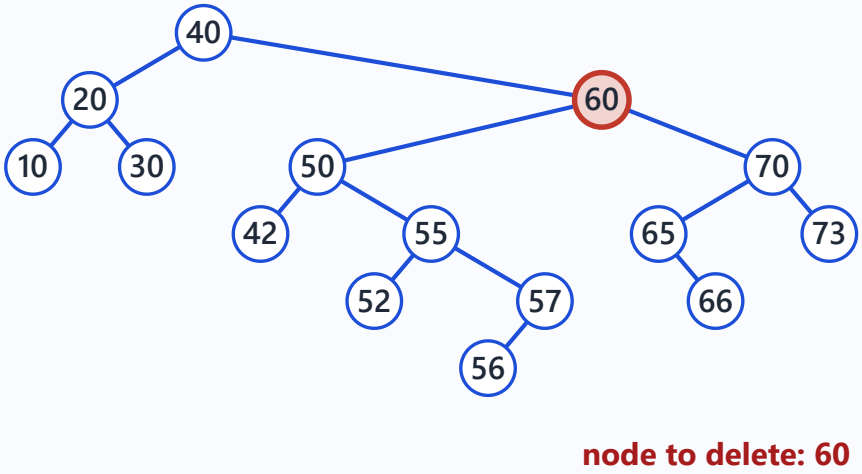


Figure 11.8: The tree used in the deletion examples. It was built by inserting 40, 20, 60, 10, 30, 50, 70, 42, 55, 65, 73, 52, 57, 66, 56 in this order; its depth is 6. The node to be deleted is 60, which has two children.

11.5.2 A node with two children: the largest of the left subtree

Which value can take the place of 60? The new value must be larger than every value in the left subtree and smaller than every value in the right subtree. There are two candidates. The first is the largest value of the left subtree. This value comes immediately before 60 in inorder; it is called the **predecessor** of 60. The second is the smallest value of the right subtree. This value comes immediately after 60 in inorder and is the **successor** of 60. When either is put in place of 60, the order is preserved. The first method of the slides uses the predecessor.

How is the largest value in the left subtree found? In a subtree every right child is larger than its parent. So the largest value is found by starting at the root of the subtree and moving along right pointers. The first node with no right child is the largest. In the example the root of the left subtree of 60 is 50. From 50 we go right to 55, and from 55 right to 57. 57 has no right child; the largest of the left subtree is 57.

The largest element has no right child, but it may have a left subtree. In the example, 56 stands to the left of 57. When 57 is taken from its place, this left child must not be forgotten. 56 is moved to the right link of 55, the parent of 57. This is safe: 56 was in the right subtree of 55 and is larger than 55.

The move can be done in two ways. In the first way, the *node* of 57 is physically moved into the place of 60. For this, four links change: the right link of 40 points to 57, the right link of 55 to 56, the left link of 57 to 50, and the right link of 57 to 70. In the second way, the node of 60 is not released with `free`. Only 57 is written into its data field. Then the old node of 57 is deleted like a node with one child. The right link of 55 is updated to 56, and this single link change is enough. The second way is shorter and the code in the slides follows it. If the node has large fields other than the data, copying data can be expensive. In that case the first way is preferred.

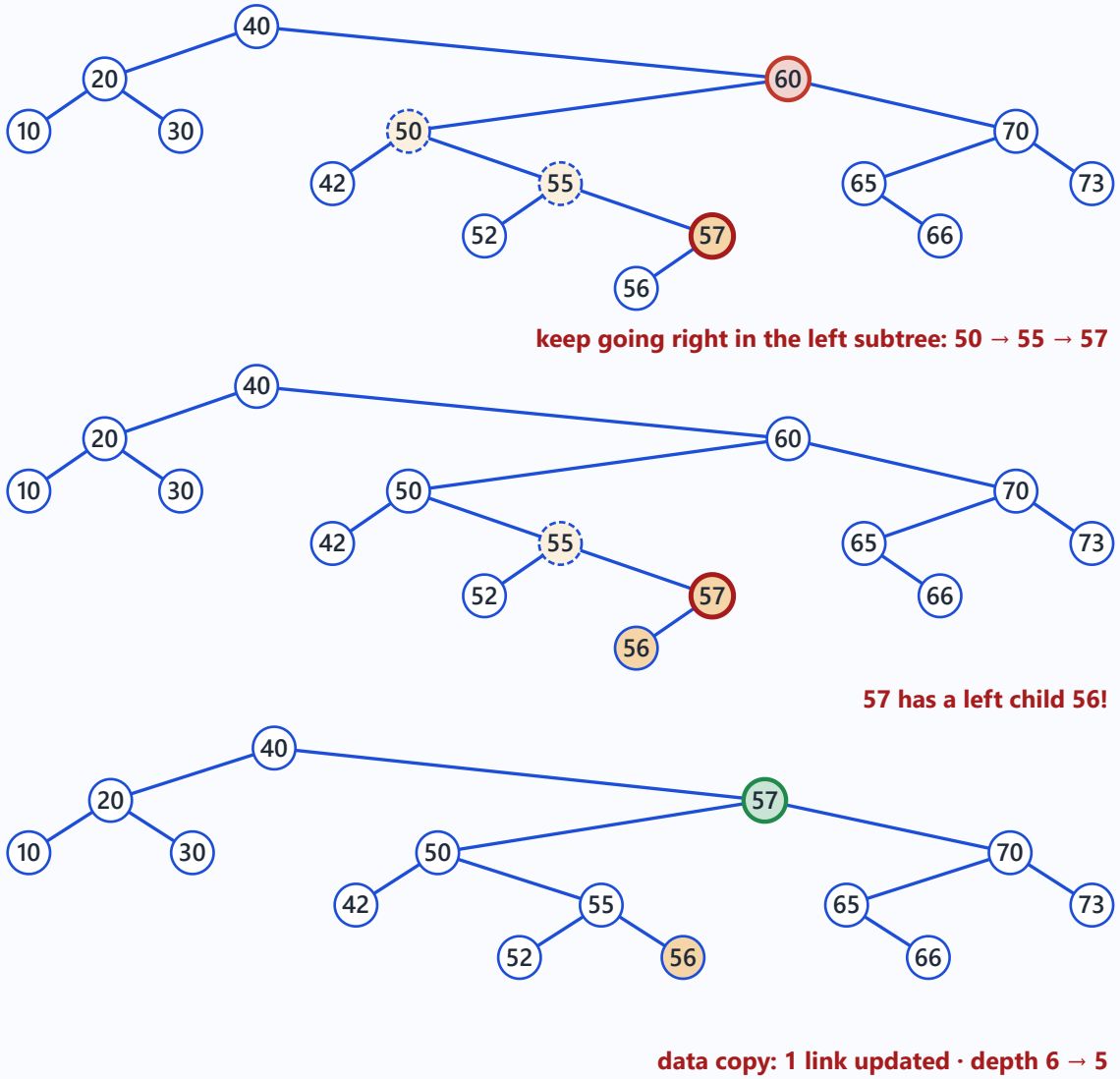


Figure 11.9: The first method. Top: the largest value is found by going right repeatedly in the left subtree: 50 → 55 → 57. Middle: the left child of 57, which is 56, is seen; 56 must be moved to the right link of 55. Bottom: the data-copy way has been applied: 57 has been written in place of 60 and only the right link of 55 has been updated. The depth of the tree has dropped from 6 to 5.

11.5.3 The deletion procedure and the pointer to a pointer

The deletion procedure also takes the tree as `tree_ptr *L`. If the deleted node is the root, the caller's root pointer must change. But the real subtlety of the procedure is that it uses the same idea not only for the root but for every link of the tree.

```
void bst_delete(int x, tree_ptr *L) {
  tree_ptr p = *L, *link = L;  // link to p
  while (p && p->data != x) {  // find x
    link = (x < p->data) ? &p->left
                         : &p->right;
    p = *link;
  }
  if (!p) return;              // x absent
  if (p->left && p->right) {  // 2 children
    tree_ptr *b = &p->left;     // left subtree's
    while ((*b)->right)           // largest
      b = &(*b)->right;           // node
    p->data = (*b)->data;      // move the data
    link = b;  p = *b;          // delete it
  }
  *link = p->left ? p->left : p->right;
  free(p);
}
```

At the center of the procedure stands the variable `link`. Its type is `tree_ptr *`, that is, a pointer to a pointer. `link` holds the *address* of the pointer that points to the node `p` currently under examination. This pointer is either the caller's root variable or the `left` or `right` field of a parent. Writing to `*link` changes the link that leads to `p`. So there is no need to keep separately who the parent is and which of its children `p` is. The pointer `q` in the insertion procedure and the "left or right?" question in its last line are merged here in a single variable.

- **Line 2.** `p` holds the root and `link` holds the address of the root pointer (`L`). At the start `*link == p`.
- **Lines 3–7.** `x` is searched for. At each step `link` is first turned into the address of the left or right field of `p`. Then we descend one level with `p = *link`. When the loop ends, either `p` is `NULL` or `p->data == x`. In both cases the equality `*link == p` is preserved.
- **Line 8.** If `x` is not in the tree, the procedure returns without doing anything.
- **Lines 9–15.** If there are two children, the largest of the left subtree is sought. `b` is also the address of a pointer. At first it points to the left field of 60; it advances along right fields until `(*b)->right` is `NULL`. When the loop ends, `*b` is the largest node and `b` is the address of the link that points to this node. Line 13 copies the largest value into the data field of the node to be deleted. Line 14 changes the node to be deleted: now the old largest node will be deleted, and the link leading to it is `b`. Since this node has no right child, it has at most one child.
- **Line 16.** At this point `p` has at most one child. The link is turned to the left child if there is one, otherwise to the right child. If there is no child at all, the right child is `NULL` and the link becomes `NULL`. A leaf, a single child, and the second stage of the two-children case are all solved by the same line.
- **Line 17.** The node detached from the tree is freed with `free`.

The order of the last two lines is important. First `*link` is updated, then `p` is released. If the order were reversed, line 16 would read the fields of a freed node. Another detail is the notation `(*b)->right` in line 11. The parentheses are necessary, because the operator `->` is applied before `*`. If `*b->right` were written, the compiler would take `b` to be a pointer to a structure and the code would not compile.

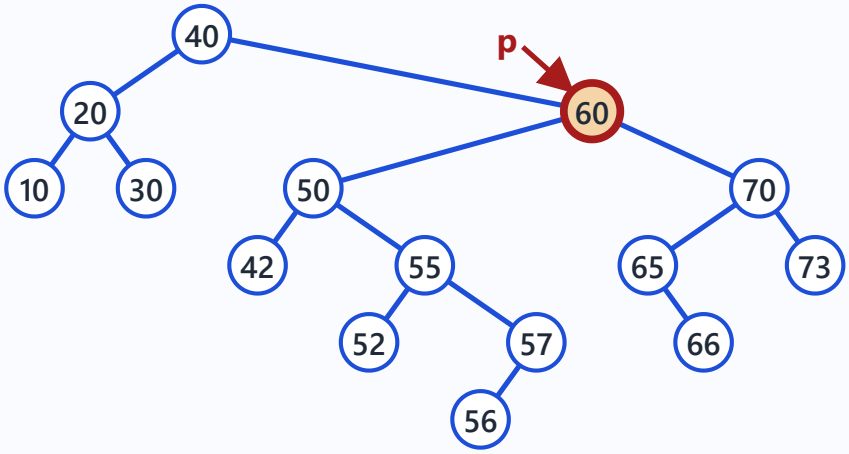
If the procedure cannot find the value to be deleted, it returns silently. After a deletion the number of nodes drops by one. This was also checked in the driver program: at every deletion the `free` counter rose by one and the `malloc` counter did not change. When all 15 values of the sample tree were deleted one by one, the root pointer became `NULL`. From a tree of 2000 random values, all the values were deleted in random order. After each deletion it was checked that the tree was still a binary search tree, and in the end the `malloc` and `free` counts came out equal.

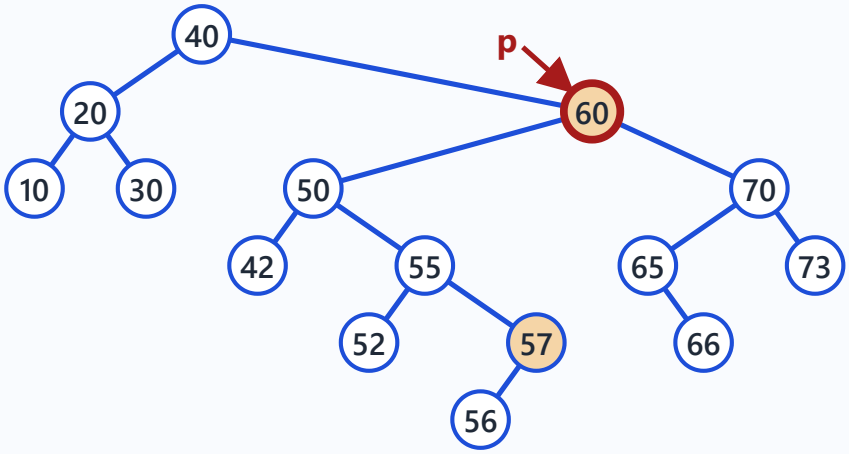
11.5.4 Code trace: deleting 60

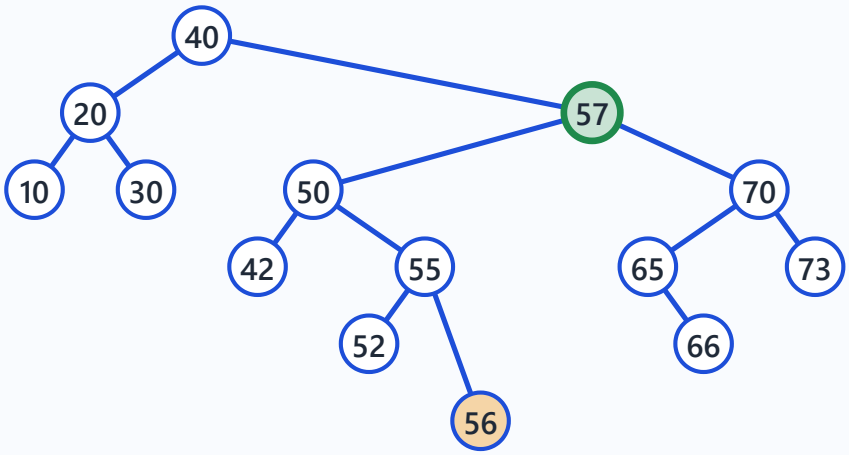
Table 11.6 traces the call `bst_delete(60, &L)` step by step. The addresses `link` and `b` are written in the form "&40->right", that is, by the field they point to.

Line	`p`	`link`	`b` / `*b`	Explanation
2	40	`L`	—	Start: `link` is the address of the root pointer.
4–6	60	&40->right	—	*60 > 40*: descend to the right; the loop ends at 60.
9–10	60	&40->right	&60->left / 50	There are two children: move to the left subtree.
11–12	60	&40->right	&50->right / 55	50 has a right child.
11–12	60	&40->right	&55->right / 57	55 has a right child.
11	60	&40->right	&55->right / 57	57 has no right child: the largest is 57.
13	60 (data 57)	&40->right	same	57 is written into the data field of 60.
14	old 57	&55->right	same	The node to be deleted is now the old 57.
16	old 57	&55->right	—	`*link = 56`: the right of 55 becomes 56.
17	—	—	—	The old node 57 is freed with `free`.

Table 11.6: Trace of the call `bst_delete(60, &L)`.


40
20
10
30
p
60
50
42
55
52
57
56
70
65
66
73


40
20
10
30
p
60
50
42
55
52
57
56
70
65
66
73


40
20
10
30
57
50
42
55
52
56
70
65
66
73

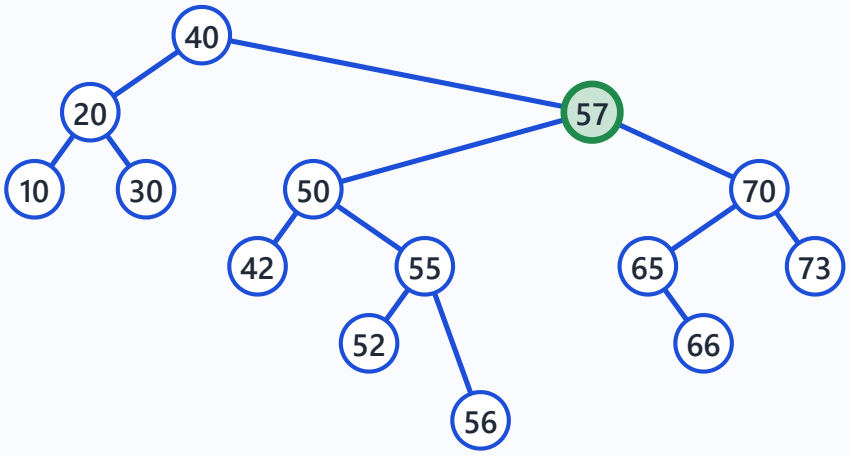


Figure 11.10: Four frames from the trace of `bst_delete(60, &L)`. Left to right and top to bottom: `link` points to the right field of 40 and `p` is at 60; `b` has reached the largest of the left subtree (57); the right link of 55 has been turned to 56; the old node has been freed and the depth has dropped to 5.

After the deletion the tree is 40(20(10, 30), 57(50(42, 55(52, 56)), 70(65(–, 66), 73))). The inorder traversal is 10 20 30 40 42 50 52 55 56 57 65 66 70 73 and is still sorted. The driver program verified this tree and that its depth is 5.

The cost of the deletion procedure is the sum of two paths: the path descending to the node to be deleted and the path from there down to the largest element. Together these cannot be longer than a single path from the root to a leaf. The total is *O(h)*: *O(lg n)* in a balanced tree and *O(n)* in a skewed tree. The procedure is iterative and uses a few pointers as extra memory: *O(1)*.

Common mistake

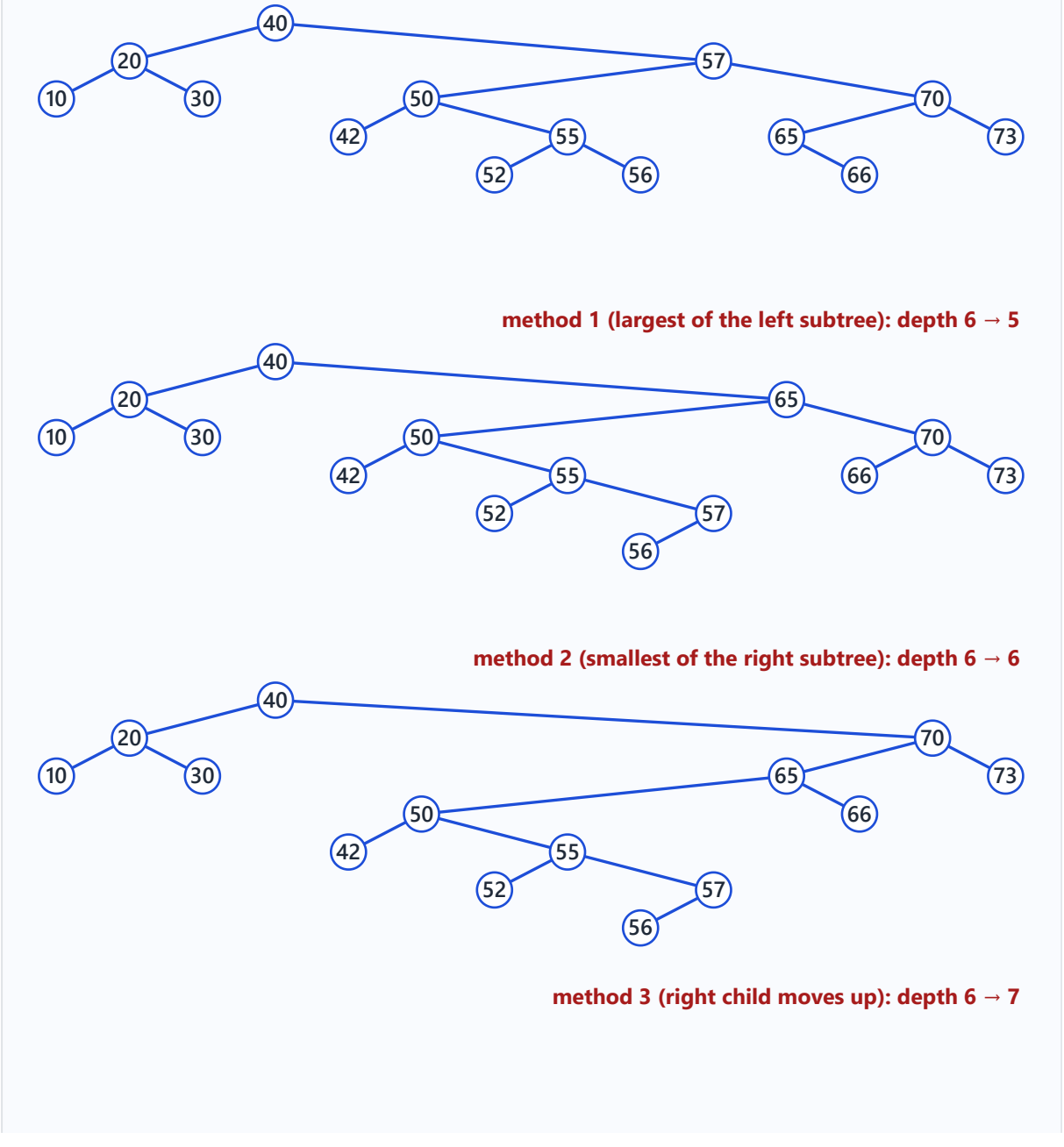
Forgetting the left child of the moved element in a two-children deletion. In the example, if the right link of 55 is set to `NULL` when 57 is taken up, 56 is cut off from the tree. The node is still in memory, but no pointer points to it: the value is lost and memory leaks. The code in the slides prevents this with line 16; it always turns the link to the child of the deleted node, if there is one.

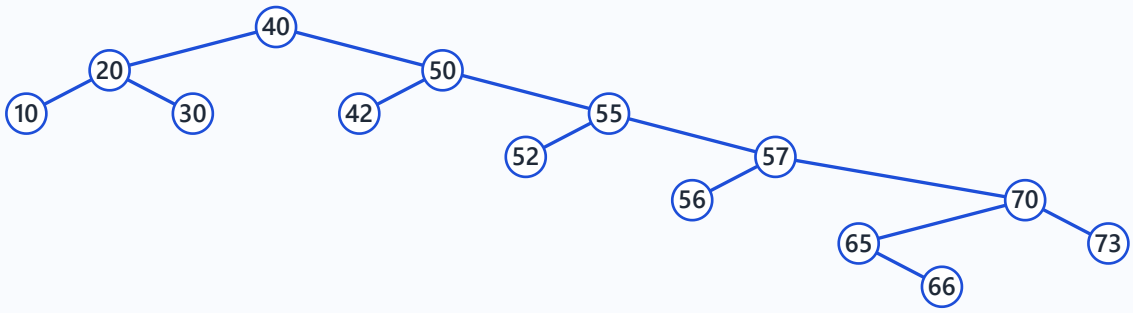
11.5.5 Four methods and their effect on depth

The slides compare four methods for deleting a node with two children. The first two put a value in place of the deleted value; the last two move a subtree up.

1. **The largest of the left subtree.** This is the method described in the previous subsection. 57 moves up and 56 is attached to the right of 55. The depth drops to 5.

2. **The smallest of the right subtree.** This is the mirror method. In the right subtree of 60 we go left repeatedly: 70 → 65. 65 has no left child; the smallest is 65. 65 is written in place of 60. The right child of 65, 66, is attached to the left of 70. Since the long arm under 50 stays in place, the depth remains 6.
3. **The right child moves up.** The right child (70) slides into the place of the deleted node. The left subtree (the subtree of 50) needs somewhere to hang. Every value in 50's subtree is smaller than every value in 70's subtree. So the subtree of 50 is hung to the left of the smallest node (65) of 70's subtree. The depth rises to 7.
4. **The left child moves up.** This is the mirror of the third. The left child (50) slides into its place. The subtree of 70 is hung to the right of the largest node (57) of 50's subtree. The path 40 → 50 → 55 → 57 → 70 → 65 → 66 is seven nodes long; the depth is 7.





method 4 (left child moves up): depth 6 → 7

Figure 11.11: The four methods of deleting 60, from top to bottom. In the first method 57 moves up and the depth drops to 5. In the second method 65 is moved, 66 is attached to the left of 70, and the depth stays 6. In the third method 70 slides up and the subtree of 50 is hung to the left of 65. In the fourth method 50 slides up and the subtree of 70 is hung to the right of 57. In the last two the depth rises to 7.

All four results are valid binary search trees. In the driver program all four were applied separately. In each, the inorder traversal gave the same sorted sequence. The depths came out as 5, 6, 7, and 7 respectively. The difference between them is in the shape of the tree and therefore in the cost of later searches. The last two methods increase the depth because they hang one subtree at the very bottom of another. The first two methods lift only a single node and do not increase the depth. The instructor therefore proposes this principle: the deletion should be done so that the tree rooted at the deleted element stays as balanced as possible. In practice the first two methods are used. Some implementations alternate between the two, so that always using the same side does not make the tree lean to one side over time.

The code for the second method is the mirror image of the first: we move into the right subtree instead of the left and follow left pointers instead of right.

```
void bst_delete_right(int x, tree_ptr *L) {    /* method 2 */
  tree_ptr p = *L, *link = L;
  while (p && p->data != x) {
    link = (x < p->data) ? &p->left
                         : &p->right;
    p = *link;
  }
  if (!p) return;
  if (p->left && p->right) {    // 2 children
    tree_ptr *b = &p->right;       // right subtree's
    while ((*b)->left)             // largest
      b = &(*b)->left;             // node
    p->data = (*b)->data;
    link = b;  p = *b;
  }
  *link = p->left ? p->left : p->right;
  free(p);
}
```

The expression in the last line is the same in both methods. The smallest node of the right subtree has no left child. So the expression turns the link to its right child. In the example, the left link of 70 points to 66.

11.6 Balanced trees

The cost of every operation in the previous sections was $O(h)$. This section shows in numbers how large h can grow. It then describes in detail the AVL tree, which keeps the height $O(\lg n)$ in every case. It also touches briefly on the red-black tree and the B-tree.

11.6.1 The cost of imbalance

Table 11.7 compares the best-case and worst-case depth of a tree with n values. The depth of a balanced tree is $\lceil \lg(n + 1) \rceil$ and that of a skewed tree is n.

n	Balanced: $\lceil \lg(n + 1) \rceil$	Skewed: n
1.000	10	1.000
1.000.000	20	1.000.000
2.000.000	21	2.000.000
10^9	30	10^9

Table 11.7: Worst-case search length (number of nodes examined) in a balanced and a skewed tree.

The difference in the table is striking. The instructor sums it up in one sentence: the difference between $O(n)$ and $O(\lg n)$ is like the difference between 2,000,000 and 21. In an array of a billion records, a balanced tree finds a record in 30 steps. A skewed tree, in the worst case, takes a billion steps.

With data arriving in random order the situation is not that bad. When every order of the values is equally likely, the average depth of a tree with n values is $\Theta(\lg n)$. The average number of comparisons of a successful search is about $2 \ln n \approx 1.39 \lg n$. This claim was measured in the driver program. 1000, 10,000, 100,000, and 1,000,000 random values were inserted into empty trees and the average level of the nodes was computed. The results were 12.1, 16.7, 21.0, and 25.2. The exact formula $2(1 + 1/n)H_n - 3$ gives 12.0, 16.6, 21.2, and 25.8 for the same sizes. Here H_n is the sum $1 + 1/2 + \ldots + 1/n$. The measurements are very close to the formula. The value $1.39 \lg n$ is the leading term of this formula and is about two more than what was measured at every size. The deepest path of the tree was longer: on average 47.7 levels at one million values. This is a little more than twice $\lg n \approx 20$.

A good average case is no guarantee. The worst case is sorted or nearly sorted data, which in fact is common. For this reason, trees have been developed that keep the depth at $O(\lg n)$ through local rotations after every insertion and deletion. The AVL tree, the red-black tree, and the B-tree are the

best known of them.

11.6.2 The AVL tree

The AVL tree is the first self-balancing binary search tree. Its name comes from the initials of G. M. Adelson-Velsky and E. M. Landis, who published the structure in 1962. In this section the **height** of a node is the depth of the subtree rooted at that node. The height of a leaf is 1 and that of the empty tree is 0. In the code this value appears under the name `ht`.

Definition 11.2

The **balance factor** of a node is found by subtracting the height of its right subtree from the height of its left subtree: *ht(left) − ht(right)*. An **AVL tree** is a binary search tree in which the balance factor of every node is −1, 0, or +1.

The condition must hold at every node, not only at the root. A node with balance factor +1 is slightly heavy on the left, a node with −1 slightly heavy on the right; a node with 0 is perfectly balanced. The following theorem shows how strong the condition is.

Theorem 11.3 (Height of an AVL tree)

The height of an AVL tree with n nodes does not exceed about $1.44 \lg(n + 2)$. Therefore search, insertion, and deletion in an AVL tree take $O(\lg n)$.

Proof sketch. Consider the smallest possible number of nodes of an AVL tree of height h; call it $N(h)$. $N(1) = 1$ and $N(2) = 2$. One subtree of the sparsest tree of height h must have height $h - 1$. The height of the other subtree is at least $h - 2$ because of the balance condition. If both subtrees are chosen as sparse as possible, we get $N(h) = N(h - 1) + N(h - 2) + 1$. This recurrence resembles that of the Fibonacci numbers. The values $N(h) + 1$ follow the Fibonacci sequence: 2, 3, 5, 8, 13, … The Fibonacci numbers grow at the rate φ^h, where $\varphi = (1 + \sqrt{5})/2 \approx 1.618$. Hence $n \geq N(h) \approx \varphi^h$, that is, $h \leq \log_{\varphi} n = \lg n / \lg \varphi \approx 1.44 \lg n$. An exact computation, with small corrections, gives the bound $1.44 \lg(n + 2)$. ■

$N(h)$ was computed in the driver program: $N(5) = 12$, $N(10) = 143$, $N(20) = 17{,}710$, $N(30) = 2{,}178{,}308$. So an AVL tree of height 30 has more than two million nodes. An AVL tree with two million nodes has 29 levels in the worst case, whereas a skewed tree could have two million levels.

The AVL node in the slides adds a height field to the ordinary node. Two helper procedures shorten the computation of height and balance factor.

```
typedef struct avl *avl_ptr;
struct avl {
    int      data;
    int      ht;        /* subtree height */
    avl_ptr left, right;
};
int ht(avl_ptr t) { return t ? t->ht : 0; }
int balance(avl_ptr t) {
    return ht(t->left) - ht(t->right);
}
```

The procedure `ht` returns 0 for the empty tree. This way the caller does not have to check separately whether the subtree is `NULL`. The procedure `balance` computes the difference in the definition; it assumes that `t` is not empty. Because the height is stored at every node, the balance factor is found in *O(1)*. If the height were not stored, the depth of the subtrees would be recomputed at every check and every operation would be *O(n)*. The node size does not change: the `int` data and the `int` height together take 8 bytes, which with two pointers makes 24 bytes in total. The 4 bytes of alignment padding in the ordinary node go to the height field here.

The general plan of insertion is as follows. The insertion is first done as in an ordinary binary search tree. Then the heights and balance factors on the path from the inserted node up to the root are updated. If an ancestor whose factor is ±2 is found, the tree is repaired at that ancestor by a local rotation. There are four cases, according to which grandchild of that ancestor the new node is under: LL, RR, LR, and RL. LL and RR are repaired with a single rotation, LR and RL with a double rotation.

11.6.3 Single rotation: LL and RR

A **rotation** is an operation that changes the local structure of the tree while preserving its order. In the figures of the slides, `z` is the first ancestor whose balance is broken. `y` is its child and `x` is its grandchild. The triangles T1–T4 are subtrees of known heights.

The LL case. The new node has entered the left subtree of the left child (`y`) of `z` . The factor of `z` rises to +2. To repair it, `z` is rotated to the right. `y` moves up and `z` becomes its right child. The right subtree of `y` , T3, moves to the left of `z` . This move is safe: the values in T3 are larger than `y` and smaller than `z` . The inorder sequence is the same before and after the rotation: T1 x T2 y T3 z T4.

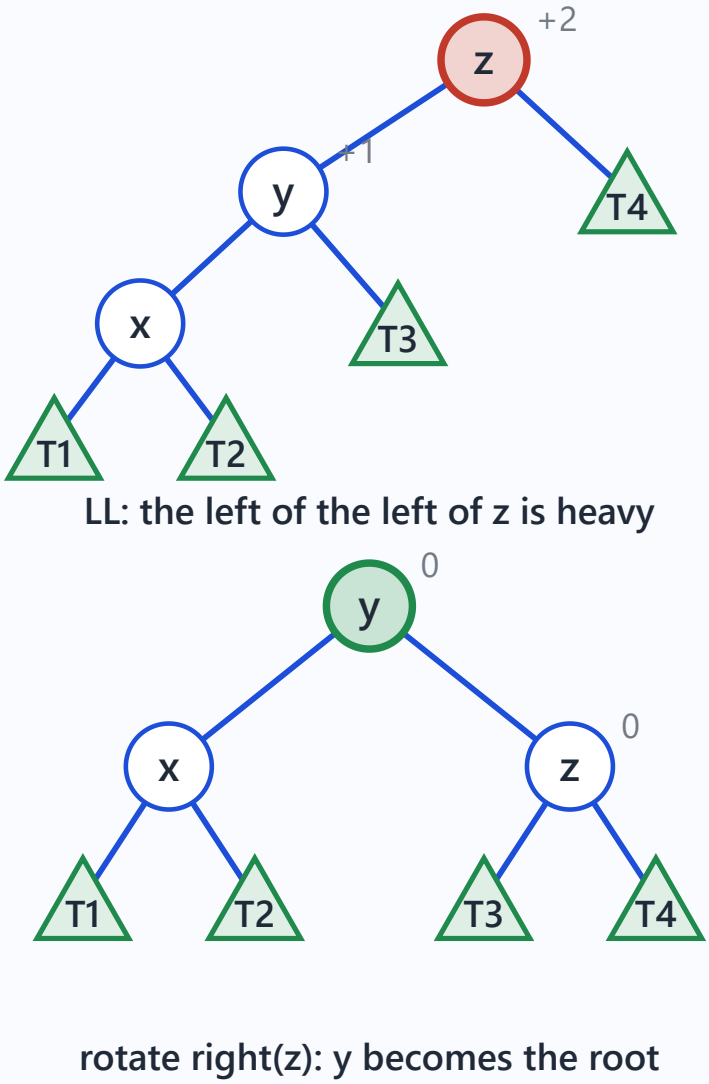


Figure 11.12: The LL case and right rotation. Left: the factor of `z` is +2; the weight is on the left of the left. Right: `z` has been rotated right, `y` has become the root of the subtree, and T3 has moved to the left of `z`.

The RR case is the mirror image of LL. The new node has entered the right subtree of the right child of `z`, and the factor of `z` is −2. `z` is rotated to the left; the left subtree of `y`, T2, moves to the right of `z`.

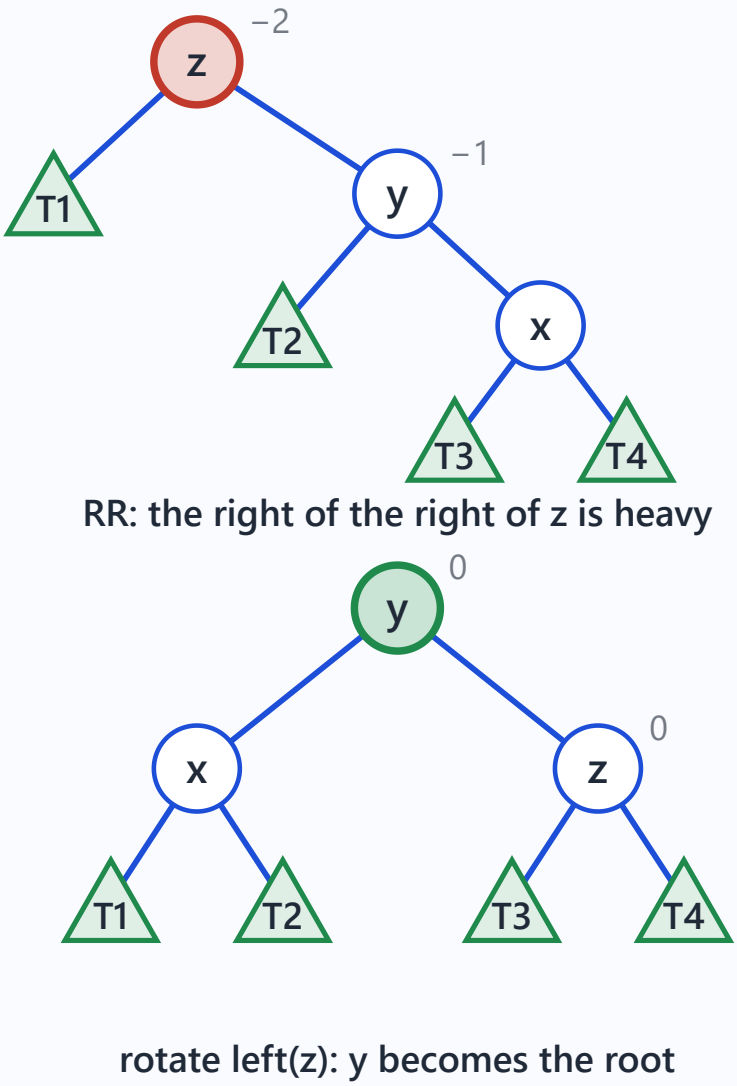
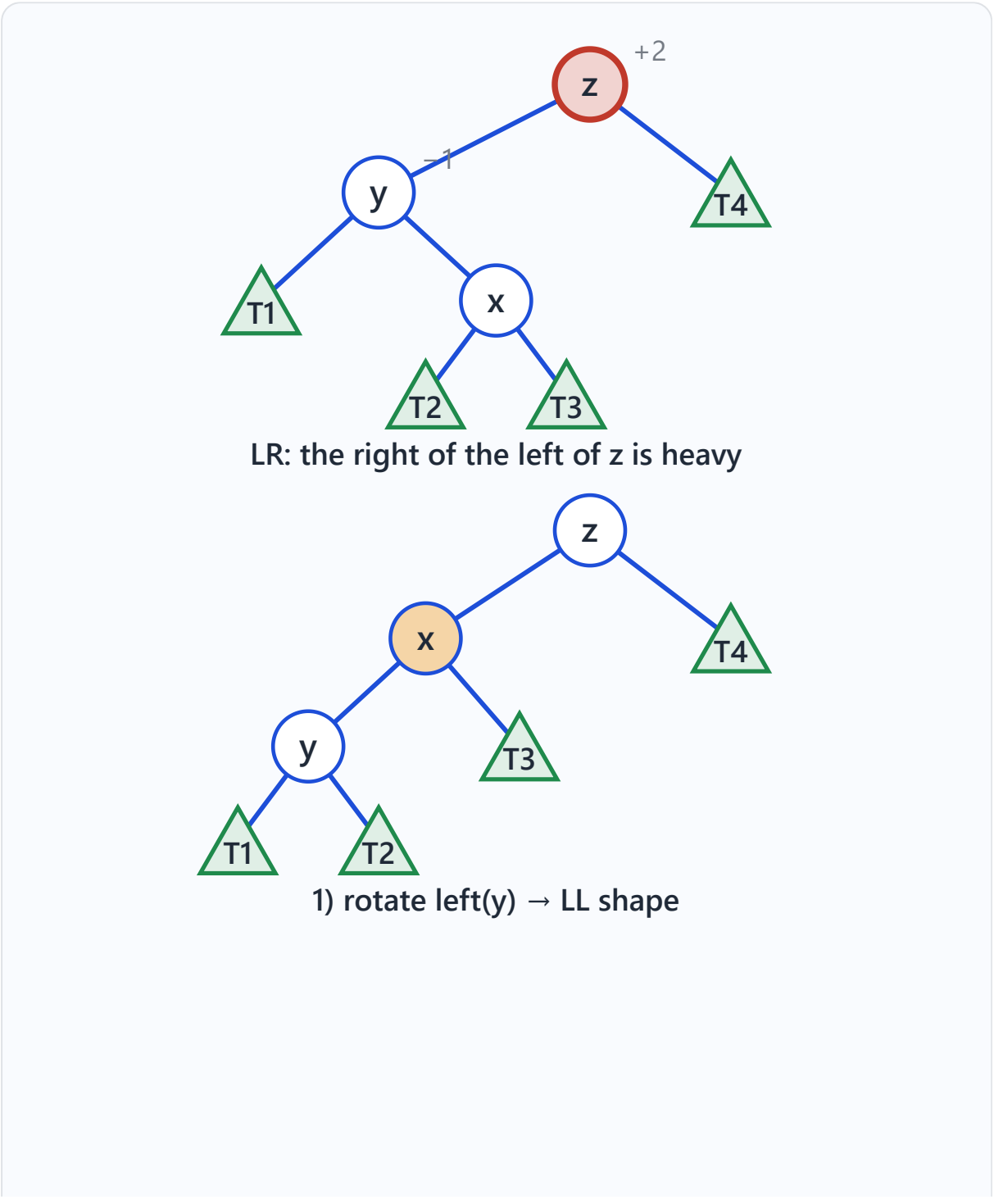


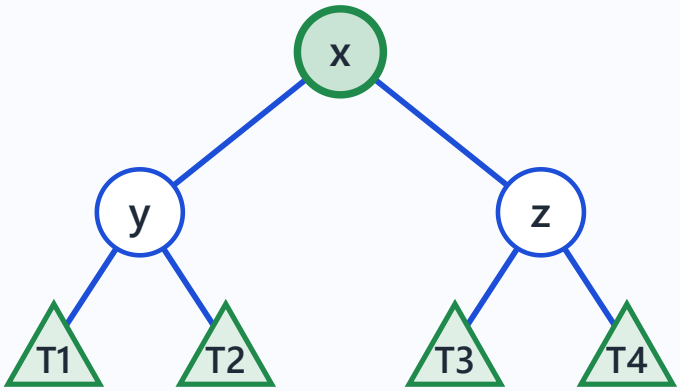
Figure 11.13: The RR case and left rotation. Left: the factor of `z` is −2; the weight is on the right of the right. Right: `z` has been rotated left, `y` has become the root, and T2 has moved to the right of `z`.

In both cases, after the rotation the height of the subtree returns to its value before the insertion. For LL this is shown as follows. Let the height of the subtree of `x` before the insertion be *k*. Since `z` was balanced and the insertion raised it to +2, before the insertion the height of T3 was *k*, that of `y` was *k + 1*, and that of T4 was *k*. The height of the subtree was *k + 2*. The insertion raises the subtree of `x` to *k + 1* and `y` to *k + 2*. After the rotation the two subtrees of `z` are T3 and T4. Since both have height *k*, the height of `z` becomes *k + 1*. Under `y` there are two subtrees of height *k + 1*: the subtree of `x` and `z`. So the height of `y` becomes *k + 2*. The height of the subtree is as it was before the insertion. Therefore the factors of the ancestors above do not change, and a single rotation suffices in insertion.

11.6.4 Double rotation: LR and RL

The LR case. The new node has entered the *right* subtree of the left child (`y`) of `z` , that is, under `x` . A single right rotation does not help here. If `z` were rotated right, the heavy subtree of `x` would move to the left of `z` and the imbalance would shift to the other side. The solution takes two steps. First `y` is rotated left. `x` takes the place of `y` , and the case turns into the LL shape. Then `z` is rotated right. The grandchild `x` rises above its two ancestors. `y` sits on its left and `z` on its right. Of the old subtrees of `x` , T2 moves to the right of `y` and T3 to the left of `z` . This operation, made of two single rotations, is called a **double rotation**.





2) rotate right(z): x is the root

Figure 11.14: The LR case and double rotation. Top: the factor of `z` is +2 and that of `y` is −1; the weight is on the right of the left. Middle: `y` has been rotated left and the case has reached the LL shape. Bottom: `z` has been rotated right and `x` has become the root.

The RL case is the mirror image of LR. The new node is in the left subtree of the right child of `z`. First the right child is rotated right, then `z` is rotated left. As a result of LR and RL, `x` is the median of the three nodes; that is why it rises to the root. In all four cases the result has the same shape: of the three nodes, the one with the median value becomes the root, the smaller becomes the left child and the larger the right child; the four subtrees are distributed below them in order.

11.6.5 Rotation code and trace

The right rotation code changes only two pointers and recomputes two heights. The naming in the slides, as in Weiss's book, is `k1` and `k2`: `k2` is the node whose balance is broken and `k1` is its left child.

```
avl_ptr rot_right(avl_ptr k2) {  // LL
  avl_ptr k1 = k2->left;
  k2->left = k1->right;
  k1->right = k2;
  k2->ht = 1 + MAX(ht(k2->left), ht(k2->right));
  k1->ht = 1 + MAX(ht(k1->left), k2->ht);
  return k1;                       // new root
}
```

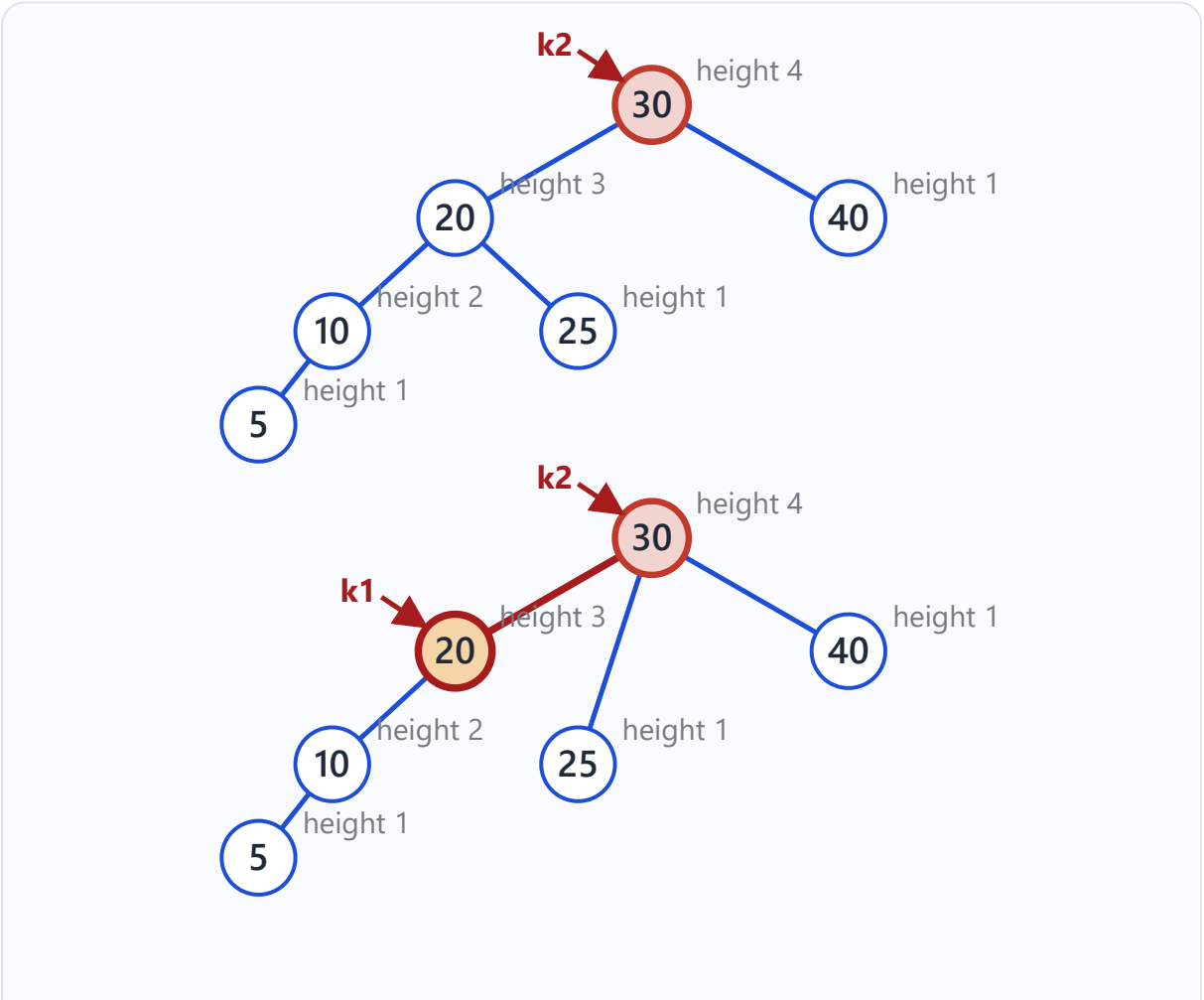
- **Line 2.** `k1` is the left child of `k2` and will be the new root.
- **Line 3.** The right subtree of `k1` moves to the left of `k2`. This is the move of T3 in the figure.
- **Line 4.** `k2` becomes the right child of `k1`.

- **Lines 5–6.** The heights are updated from the bottom up. First the height of `k2` and then that of `k1` is computed. The new right child of `k1` is `k2`, and its height was found in the previous line. If the order is reversed, `k1` uses an old, wrong value.
- **Line 7.** The new root is returned. The caller attaches the parent's pointer to this node.

The slides trace the procedure with a small example. In the tree, 30 is the root, with 20 on its left and 40 on its right. The children of 20 are 10 and 25. When 5 is inserted, it goes to the left of 10. The factor of 30 becomes *ht(20) − ht(40) = 3 − 1 = +2*. The new value is on the left of the left; the case is LL and `rot_right(30)` is called.

Line	Operation	Result
2	`k1 = k2->left`	`k1` = 20
3	`k2->left = k1->right`	25 moves to the left of 30 (*20 < 25 < 30*); for now two pointers point to 25.
4	`k1->right = k2`	30 becomes the right child of 20.
5	`k2->ht`	*1 + MAX(ht(25) = 1, ht(40) = 1) = 2*
6	`k1->ht`	*1 + MAX(ht(10) = 2, 2) = 3*
7	`return k1`	The new root is 20; the tree is 20(10(5, –), 30(25, 40)).

Table 11.8: Trace of the call `rot_right(30)` (after 5 has been inserted).



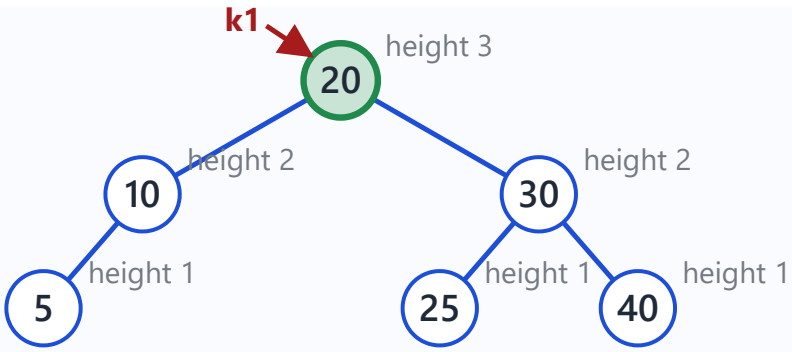


Figure 11.15: Three frames from the `rot_right` trace. Top: 5 has been inserted and the factor of 30 is +2. Middle: 25 has been moved to the left of 30 and 30 has been attached to the right of 20; the heights have not been updated yet. Bottom: the heights have been corrected and 20 is the root.

After the rotation the height of the subtree is 3; it was 3 before 5 was inserted too. Only two pointers changed. The rotation does a constant amount of work and its cost is *O(1)*. In the driver program the same insertion sequence (30, 20, 40, 10, 25, 5) was applied. The single rotation was `rot_right(30)` and the resulting tree came out as in the table.

The slides do not give the code of the left rotation, but the AVL insertion code uses it. The left rotation is the mirror image of the right rotation: `left` and `right` are exchanged. The version used in the driver programs and the procedure that allocates a new AVL node are below.

```
avl_ptr rot_left(avl_ptr k1) {  // RR
  avl_ptr k2 = k1->right;
  k1->right = k2->left;
  k2->left = k1;
  k1->ht = 1 + MAX(ht(k1->left), ht(k1->right));
  k2->ht = 1 + MAX(k1->ht, ht(k2->right));
  return k2;                        // new root
}
avl_ptr new_avl(int x) {
  avl_ptr t = malloc(sizeof *t);
  if (!t) { fprintf(stderr, "Out of memory\n"); exit(EXIT_FAILURE); }
  t->data = x;  t->ht = 1;
  t->left = t->right = NULL;
  return t;
}
```

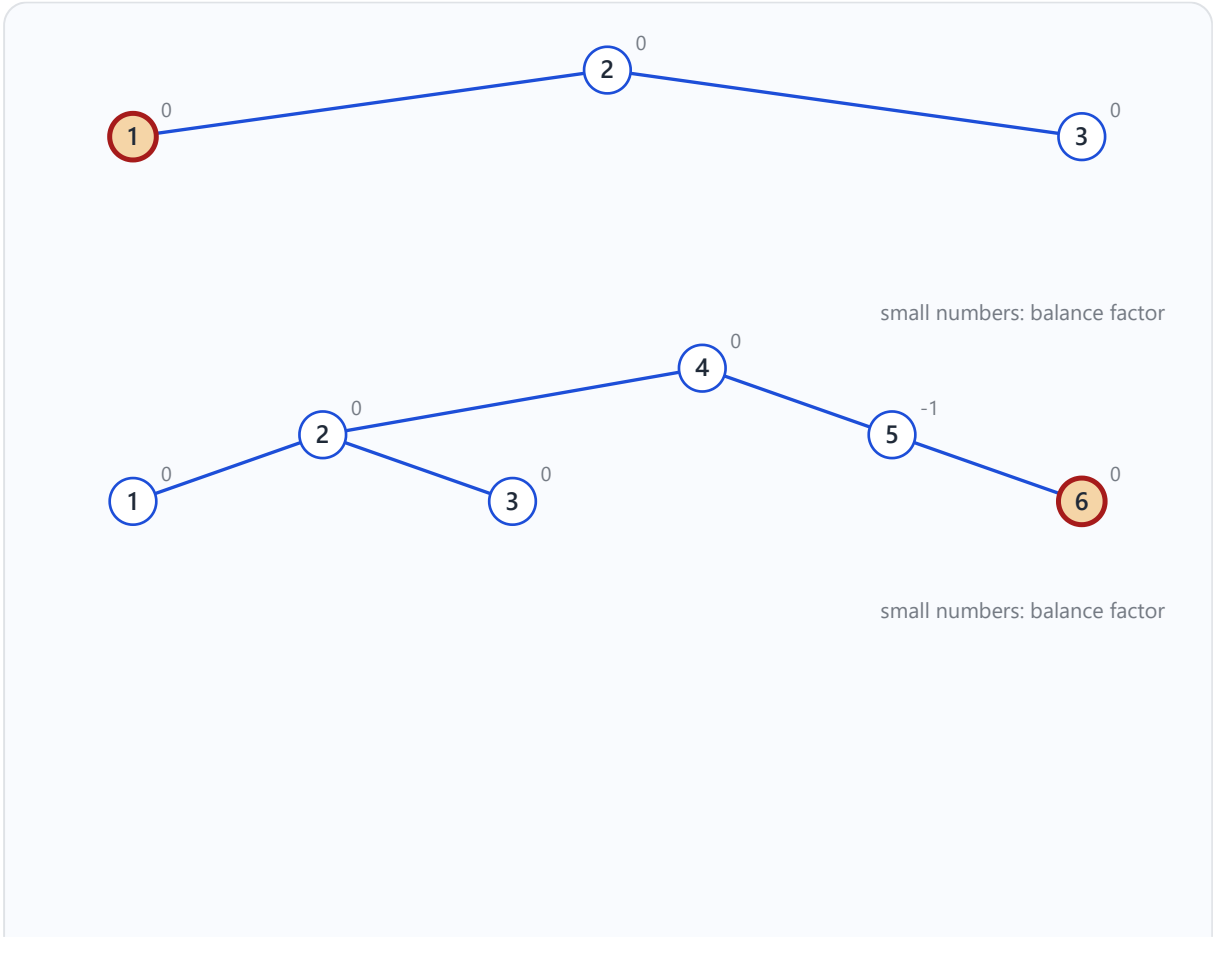
11.6.6 Insertion into an AVL tree

The slides follow the example in Weiss's book. The values 3, 2, 1, 4, 5, 6, 7, 16, 15 are inserted in this order into an empty AVL tree. Table 11.9 gives the state after each insertion.

Inserted	Case	Rotation	Result
3	—	none	3
2	—	none	3(2, –)
1	LL, +2 at 3	right at 3	2(1, 3)
4	—	none	2(1, 3(–, 4))
5	RR, −2 at 3	left at 3	2(1, 4(3, 5))
6	RR, −2 at 2	left at 2	4(2(1, 3), 5(–, 6))
7	RR, −2 at 5	left at 5	4(2(1, 3), 6(5, 7))
16	—	none	4(2(1, 3), 6(5, 7(–, 16)))
15	RL, −2 at 7	right at 16, left at 7	4(2(1, 3), 6(5, 15(7, 16)))

Table 11.9: Inserting 3, 2, 1, 4, 5, 6, 7, 16, 15 into an empty AVL tree. Trees are written in the form root(left, right); "–" is an empty subtree.

Three steps in the table deserve a separate explanation. When 6 is inserted, the imbalance appears not at the parent of the new node but at the root. The factor of 5 is −1 and that of 4 is −1; but the left subtree of the root 2 has height 1 and the right has height 3. The rotation is done at the root and 4 becomes the new root. When 7 is inserted, the imbalance is at 5 and the root is not affected. When 15 is inserted, the new node goes to the left of the right child (16) of 7. This is the RL case. First 16 is rotated right, then 7 is rotated left, and 15 becomes the parent of 7 and 16.



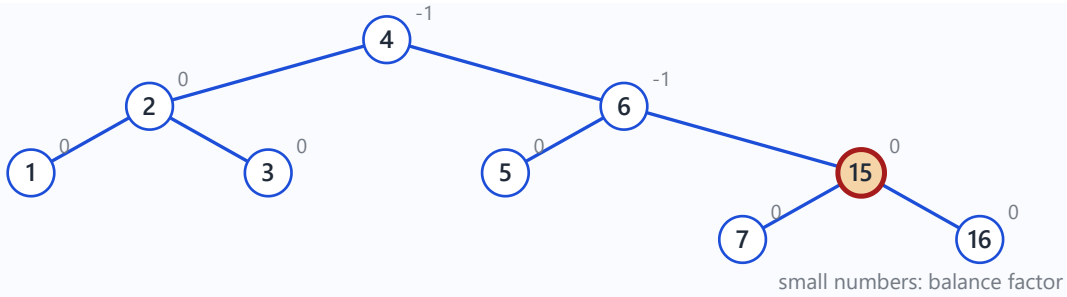


Figure 11.16: Three frames from the AVL insertion example. The small numbers beside the nodes are balance factors. Top: 1 has been inserted and the LL case has been repaired by a right rotation at 3. Middle: 6 has been inserted and an RR rotation has been made at the root. Bottom: 15 has been inserted and the final tree has been obtained by an RL double rotation.

The height of the final tree is 4. If the same sequence had been inserted into an ordinary binary search tree, the chain 3 → 4 → 5 → 6 → 7 → 16 → 15 would have formed and the depth would have been 7. After every insertion the driver program printed the tree, the factors, and the rotations performed. The results came out the same as each row of the table. It was also checked at every step that the stored heights equal the real heights and that all factors stay in the range −1..+1.

The insertion code in the slides adds a balance check to the return path of recursive binary search tree insertion.

```
avl_ptr avl_insert(avl_ptr t, int x) {
  if (!t) return new_avl(x);              /* malloc, ht = 1         */
  if (x < t->data)      t->left = avl_insert(t->left, x);
  else if (x > t->data) t->right = avl_insert(t->right, x);
  else return t;                           /* no duplicate keys      */
  t->ht = 1 + MAX(ht(t->left), ht(t->right));
  int d = balance(t);
  if (d >  1 && x < t->left->data) return rot_right(t);           /* LL */
  if (d < -1 && x > t->right->data) return rot_left(t);           /* RR */
  if (d >  1) { t->left = rot_left(t->left); return rot_right(t); } /* LR */
  if (d < -1) { t->right = rot_right(t->right); return rot_left(t); } /* RL */
  return t;
}
```

- **Lines 2–5.** This is ordinary recursive insertion. When an empty subtree is reached, a new node is allocated and its height is set to 1. If an equal value arrives, the tree is returned unchanged.
- **Lines 6–7.** While returning from the recursion, the height of this node is updated and its balance factor is computed. These lines run in turn at each ancestor on the path from the new node to the root.
- **Line 8 (LL).** If the factor is greater than +1, the weight is on the left. If `x` is smaller than the value of the left child, the new node is on the left of the left; a single right rotation suffices.

- **Line 9 (RR).** The mirror: if the factor is less than −1 and `x` is larger than the right child, a single left rotation is made.
- **Line 10 (LR).** The factor is +2 but the new node is on the right of the left. First the left child is rotated left, then the node is rotated right.
- **Line 11 (RL).** The mirror.
- **Line 12.** If the balance has not been broken, the node is returned as it is.

The code tells the cases apart by comparing the new value with the value of the child. If the new value is smaller than the left child, it has entered the left of the left; if larger, the right of the left. Since the values are unique, there is no equality. The usage is `root = avl_insert(root, x);`. Since the new root is given as the return value, no `*L` parameter is needed.

At most one rotation (single or double) suffices in insertion. The rotation brings the height of the subtree back to its value before the insertion; the ancestors above see no change. The code still computes heights and checks balance at all the ancestors on the path, but these are constant operations. The total cost is the length of the path, that is, $O(\lg n)$. The recursion keeps $O(\lg n)$ frames on the stack. Deletion is done in a similar way, but a rotation can decrease the height of the subtree. This decrease can upset the balance of an ancestor higher up. In deletion, therefore, up to $O(\lg n)$ rotations may be needed along the way to the root.

The behavior of the AVL tree on large inputs was also measured in the driver program. When the sorted values 1, 2, …, n were inserted, the height came out exactly $\lfloor \lg(n + 1) \rfloor$: 10 for 1000, 20 for a million, and 21 for two million. Because the weight always falls on the right in sorted data, only left rotations were made: 999,980 rotations in a million insertions. When a million random values were inserted, the height came out as 24. The upper bound in the theorem for this size is 28. If the same sorted data had been inserted into an ordinary binary search tree, the depth would have been a million.

11.6.7 The red-black tree and the B-tree

The AVL tree keeps balance tight. The price of this tightness is that deletion may need many rotations. Two other balanced trees are also widely used in practice.

The **red-black tree** gives each node a color, red or black, and puts rules on the colors:

- Every node is red or black; the root and the `NULL` leaves are black.
- The children of a red node are black; there are no two red nodes in a row.
- On every path from a node down to the `NULL`s below it there is the same number of black nodes.

These rules ensure that the longest path is no more than twice the shortest path. As a result the height is at most $2 \lg(n + 1)$. The balance is looser than in AVL. In return, the number of rotations in insertion and deletion is bounded. At most two rotations in insertion and at most three in deletion,

plus some recoloring, suffice. Because the structure of the tree changes less, the red-black tree is preferred in structures that are updated a lot. `std::map` in C++, `TreeMap` in Java, and many structures in the Linux kernel are red-black trees.

The **B-tree** is an *m*-way balanced search tree proposed in 1972 by R. Bayer and E. McCreight. A node holds several keys and at most *m* children. Every node other than the root holds between *⌈m/2⌉ − 1* and *m − 1* keys. All leaves are at the same level. When a node becomes full, it splits in two by moving its middle key up to its parent. When the root splits, a new root is created. So a B-tree grows from the root, not from the leaves.

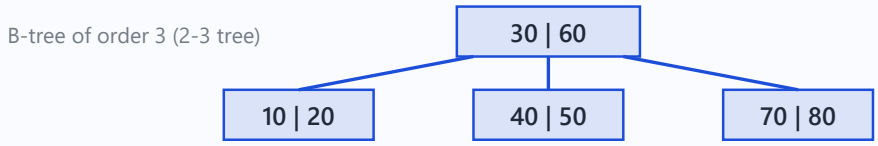


Figure 11.17: A small B-tree. Each node holds several sorted keys; one descends to the children through the gaps between the keys. All leaves are at the same level.

The main use of the B-tree is on disks and similar slow memories. Data is read from disk in blocks, and reading a block takes longer than making thousands of comparisons in memory. In a B-tree a node is matched to a disk block. *m* can be in the hundreds. The height of the tree then drops to about $\log_m n$. An index of a billion records is searched with three or four block reads. Database indexes and file systems use the **B+ tree**, a variant in which the data is kept only in the leaves.

11.6.8 Which search tree?

Table 11.10 compares the search trees mentioned in this section.

Structure	Search / insert / delete (worst case)	Balance condition	Where used
binary search tree	*O(n)* (average *O(lg n)*)	none	small, random data; teaching
AVL tree	*O(lg n)*	*|ht(left) − ht(right)| ≤ 1*	search-heavy workloads
red-black tree	*O(lg n)*	equal black height	library maps (`map`)
B-tree / B+ tree	$O(\log_m n)$ disk accesses	leaves at the same level	databases, file systems
splay tree	*O(lg n)* amortized	none; the accessed node moves to the root	frequently accessed items

Table 11.10: Comparison of search trees.

The choice depends on the nature of the job. If the data is small and in random order, the ordinary binary search tree is enough and has the simplest code. If the number of searches is much greater than the number of updates, the tighter balance of the AVL tree shortens the searches. If insertions and deletions are frequent, the few rotations of the red-black tree stand out. If the data does not fit in

memory, the B-tree is almost the only choice. If a small part of the accesses concentrates on a few values, the splay tree keeps those values near the root. The word "amortized" in the splay tree row of the table describes the average cost not of a single operation but of a long sequence of operations.

11.7 Exam questions

The last part of the slides solves three exam questions asked in previous years. All three measure the same skill: writing correct and efficient code by combining small, well-defined procedures, and computing how much work a given piece of code does.

11.7.1 The weight of an element

The question is this. In a binary tree with distinct values (it need not be a search tree), the **weight** of the element containing the value `x` is the longest of the paths from that element to the leaf nodes. If `x` is not in the tree or the tree is empty, the weight is 0. The procedure `int FindWeight(tree_ptr L, int x)` must be written without using global variables.

Read carefully, the definition shows that the weight is the depth of the subtree rooted at the node containing `x`. The problem is a composition of two well-known jobs: first find the node, then compute its depth. Since the tree is not sorted, the search is done with the procedure that works on any binary tree.

```
int FindWeight(tree_ptr L, int x) {
  /* tree_search: node holding x, or NULL */
  tree_ptr tree_search(tree_ptr L, int x);
  int depth(tree_ptr L);
  return depth(tree_search(L, x));
}
tree_ptr tree_search(tree_ptr L, int x) {
  if (!L) return NULL;
  if (L->data == x) return L;
  tree_ptr T = tree_search(L->left, x);
  return T ? T : tree_search(L->right, x);
}
int depth(tree_ptr L) {
  int max(int a, int b);  /* not smaller than a, b */
  return L ? max(depth(L->left),
                 depth(L->right)) + 1 : 0;
}
```

The beauty of the solution is in the third line: `depth(tree_search(L, x))`. If the value is not found, `tree_search` returns `NULL`. `depth(NULL)` is already 0. So the "not found" case is not handled separately; the definitions of the two procedures solve it automatically. The case `L == NULL` gives 0 the same way.

The first two lines of the code are **local prototypes** declared inside the procedure. These declarations make the names `tree_search` and `depth` visible only inside `FindWeight`. They are necessary because the procedures are defined in the file after `FindWeight`. In the same way, `depth` declares the function `max`. Using a function rather than a macro here is deliberate. A macro would make the number of calls of the depth procedure exponential.

The cost is the sum of two parts. The search examines all nodes in the worst case: $O(n)$. The depth traverses only the subtree found; that has at most n nodes. The total is $O(n)$. In the driver program, on a tree with root 2 and children 10 and 20, the weights came out as 2, 1, 1, and 0 for `x = 30`. On the tree from the deletion examples, the weight of 40 was found to be 6, of 60 it was 5, of 50 it was 4, and of 56 it was 1.

11.7.2 How many times is the procedure called?

The second question examines a bad piece of code that tries to solve the same problem. In the code, `weight(L)` is a procedure that gives the depth of the subtree rooted at `L`.

```
int FindWeight(item_ptr L, int x) {
  if (!L) return 0;
  else if (L->data == x) return weight(L);
  if (FindWeight(L->left, x) >
      FindWeight(L->right, x))
    FindWeight(L->left, x);
  if (FindWeight(L->right, x) >
      FindWeight(L->left, x))
    FindWeight(L->right, x);
  return 0;
}
```

The question has two parts. The tree has three nodes: root 2, left child 10, and right child 20. The first part asks how many times the procedure is called for `x = 20` and `x = 30`. In the second part, a recurrence for the worst case must be written and solved.

First consider what a leaf, say 10, does when it does not contain `x`. The leaf is not `NULL` and its value is not equal to `x`. The first `if` makes two recursive calls, and the second `if` makes two more. All four are made with `NULL` and return 0. Since $0 > 0$ is false, the calls inside the conditions are not made. A leaf call, together with itself, makes $1 + 4 = 5$ calls.

`x = 30` is not in the tree. The root call calls 10 twice and 20 twice. Each returns 0 and the conditional calls are not made. The total is $1 + 4 \cdot 5 = 21$ calls. For `x = 20` the situation is different. The call made for 20 finds equality in the second line and returns the weight (1) in a single call. At the root, the first `if` makes 5 calls for 10 and 1 call for 20; $0 > 1$ is false. The second `if` again makes 1 + 5 calls and $1 > 0$ is true. The `FindWeight(L->right, x)` inside the condition makes one more call.

The total is $1 + 6 + 6 + 1 = 14$. Despite all this effort, the procedure returns 0 in both cases. The results of the calls inside the conditions are not used, and the last line is always 0. The code is both slow and wrong: it gives the right answer only when `x` is at the root.

In the worst case `x` is not in the tree and the whole tree is traversed. Each call calls each of its subtrees twice. In a skewed tree one subtree has $n - 1$ nodes and the other is empty. The recurrence is $T(n) = 2T(n - 1) + c$. Unrolled, it gives $T(n) = 2^n T(0) + c(2^n - 1)$, that is, $\Theta(2^n)$. In a balanced tree each of the two subtrees has about $n/2$ nodes and all four calls are made: $T(n) = 4T(n/2) + c$. By the master theorem or by unrolling, the solution is $\Theta(n^2)$. At each level the number of calls quadruples while the problem halves, and $4^{\lg n} = n^2$.

Tree	n	Number of calls	Ratio
chain	4	61	calls / 2^n = 3.81
chain	12	16.381	3,999
chain	20	4.194.301	4,000
full	255	87.381	calls / n^2 = 1.344
full	4.095	22.369.621	1,334

Table 11.11: Number of calls made by the bad `FindWeight` procedure when `x` is not in the tree (measured with a counter).

The measurements confirm the recurrences. In a chain the number of calls is exactly $2^{n+2} - 3$ and the ratio approaches 4. In a perfect tree the ratio approaches $4/3$. The right design is the solution in the previous subsection, which takes $O(n)$. On a chain of 20 nodes the bad procedure makes more than four million calls. The right procedure makes do with a few dozen calls to look at the 20 nodes of the same tree.

The lesson of the question is the cost of computing the result of a recursive call twice. At every level the work doubles and the factors multiply with each other. If the result of a recursive call is needed more than once, it should be stored in a local variable. The `left` and `right` variables in the depth procedure do exactly this job.

11.7.3 SWAP between two calls

The third question asks about the effect of a procedure on a tree. The macro `SWAP(x, y, t)` exchanges the values of `x` and `y` by means of `t`: `((t) = (x), (x) = (y), (y) = (t))`.

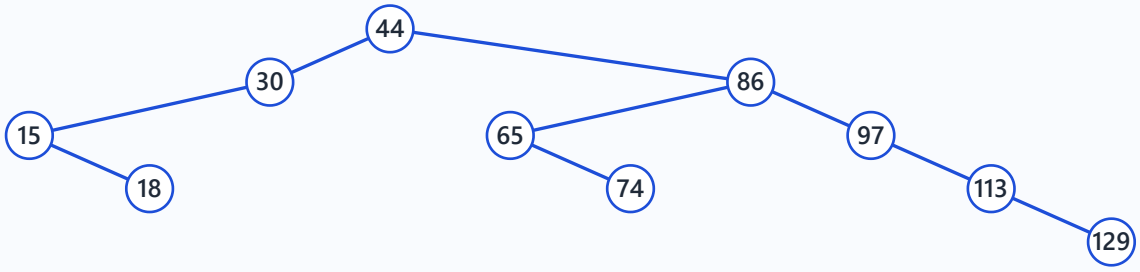
```
void proc(tree_ptr L) {
  tree_ptr p;
  if (L) {
    proc(L->left);
    SWAP(L->left, L->right, p);
    proc(L->right);
  }
}
```

The question is this: run the procedure on the binary search tree built with the values {44, 30, 15, 18, 86, 97, 65, 113, 74, 129}. If the picture of the tree changes each time the SWAP line is passed, redraw the tree and interpret its final form.

In the initial tree the root is 44. On its left is 30; to the left of 30 is 15, and to the right of 15 stands 18. To the right of the root is 86. To the left of 86 is 65 with 74 on its right; to the right is a chain extending to the right made of 97, 113, and 129.

At a node, the procedure first processes the left subtree, then exchanges the two children, then processes the *new* right child. But the new right child is the old left child, which has just been processed. The old right child has moved to the left and is never processed. The steps are as follows.

1. `proc(44)` → `proc(30)` → `proc(15)`. The left of 15 is empty. **SWAP 15**: 18 moves to the left.
2. The new right of 15 is empty. Return to 30. **SWAP 30**: the subtree of 15 moves to the right.
3. 15, the new right child of 30, is processed again. Its left (18) is processed first; SWAP is executed at 18, but since both children are empty the picture does not change. **SWAP 15**: 18 moves to the right again. The 18 on the right is processed once more; the picture again does not change.
4. Return to 44. **SWAP 44**: the subtree of 86 moves to the left and the subtree of 30 to the right.
5. 30, the new right child of 44, is processed again. Its left is empty. **SWAP 30**: the subtree of 15 turns back to the left. The new right is empty and the procedure ends.



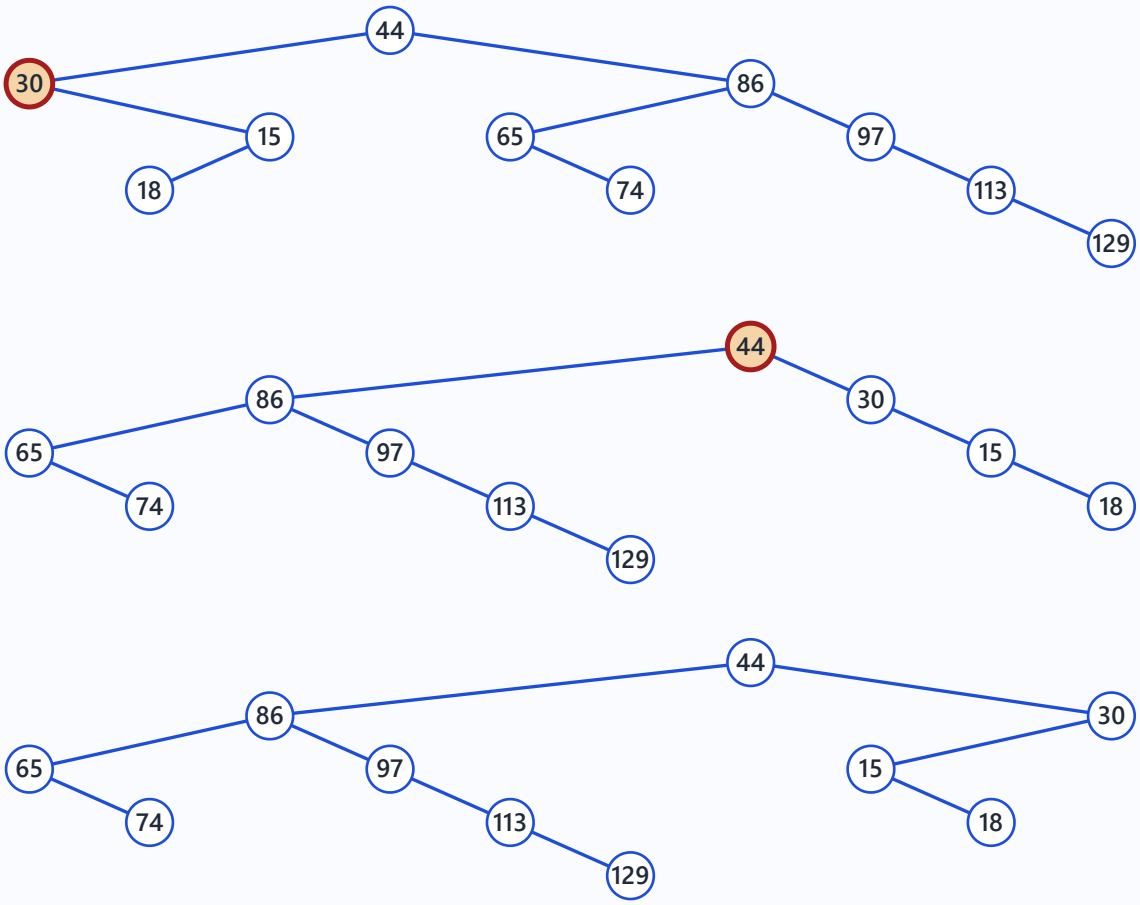


Figure 11.18: SWAP between two recursive calls. From top to bottom: the initial tree; after the first SWAP at 30; after the SWAP at 44; the final form. The two subtrees of the root have been exchanged, but the inside of the subtrees is as at the start.

The SWAPs at which the picture changes are, in order, at 15, 30, 15, 44, and 30. The two SWAPs at 18 do not change the picture. The final tree is 44(86(65(–, 74), 97(–, 113(–, 129))), 30(15(–, 18), –)). Only the two subtrees of the root have been exchanged. The 30 subtree was processed twice, and each change was undone in the second pass. The 86 subtree was not processed at all. The result is neither the tree itself nor its mirror, and it is no longer a binary search tree. The driver program printed every step by adding a trace call to the SWAP line; the order and the final tree are as above.

If SWAP had been written before or after both recursive calls, an exact mirror of the tree would have been obtained. When it is written first, the children are exchanged and then the two subtrees are mirrored separately. When it is written last, the two subtrees are mirrored first and then their places are exchanged. In both cases every subtree is processed exactly once. In a mirrored binary search tree, inorder traversal gives the values in decreasing order.

11.8 Complexity summary

Table 11.12 gives the costs of all the procedures written in these notes together. *h* is the depth of the tree; it is *O(lg n)* in a balanced tree and *n* in a skewed tree.

Operation	Best / balanced	Worst (skewed)	Extra memory
`bst_search` (iterative)	*O(lg n)*	*O(n)*	*O(1)*
`search` (recursive)	*O(lg n)*	*O(n)*	*O(h)* stack
`bst_insert`	*Ω(lg n)*	*O(n)*	*O(1)*
`bst_delete`	*O(lg n)*	*O(n)*	*O(1)*
`depth`	*Θ(n)*	*Θ(n)*	*O(h)*
search in a general binary tree	*O(n)*	*O(n)*	*O(h)*
AVL search / insert / delete	*O(lg n)*	*O(lg n)*	*O(lg n)* recursion
single rotation	*O(1)*	*O(1)*	—

Table 11.12: Time and extra-memory cost of search tree operations.

The essence of the table: in a binary search tree, search, insertion, and deletion follow a path and take *O(h)*. Operations that must look at every node of the tree (depth, search in a general tree) are *O(n)*. Balanced trees keep *h* at *O(lg n)*, making the worst case of the path-following operations logarithmic as well. The *Ω(lg n)* in the insertion row says that insertion cannot be faster than *lg n* in any tree.

11.9 Common mistakes

1. **Failing to update the root pointer.** If the insertion procedure takes `tree_ptr L`, the first insertion into an empty tree does not change the caller's root. The tree stays empty and the node leaks. Either the address must be passed with `tree_ptr *L`, or the new root must be returned and assigned.
2. **Not keeping a trailing pointer.** In iterative insertion, when `p` reaches `NULL` the parent to which the new node will be attached is no longer known. Either `q`, trailing one step behind, or a pointer to a pointer must be used.
3. **Not returning the result of the recursive call.** If only `search(...)` is written in recursive search instead of `return search(...)`, the address found is lost.
4. **Forgetting the child of the moved element.** In a two-children deletion, if the left child of the largest element of the left subtree (56 in the example) is not attached to its parent, it is cut off from the tree.
5. **Global variables and unnecessary extra parameters.** A global counter for depth, or a parameter of the "pass 0 in the first call" kind, makes the procedure fragile and not reusable.

6. **Passing a recursive call to a macro.** `MAX(depth(...), depth(...))` runs one of the calls twice and makes the time exponential.
7. **Using memory after freeing it.** In deletion, if the node is freed with `free` before the link is updated, the next line reads freed memory.
8. **Checking only against children.** Whether a tree is a binary search tree cannot be told from parent–child comparisons alone; the range coming down from the root must be applied at every node.

11.10 Exercises and solutions

This section solves the six exercises in the slides. The solutions with code were compiled and run with driver programs.

1. Insertion and preorder. Insert 50, 30, 70, 20, 40, 60, 80, 35 in this order into an empty tree. 50 becomes the root. 30 and 70 are its children. 20 and 40 are children of 30; 60 and 80 are children of 70. 35 follows the path 50 → 30 → 40 and goes to the left of 40. The tree is 50(30(20, 40(35, –)), 70(60, 80)) and its depth is 4. The preorder is 50 30 20 40 35 70 60 80.

2. Deleting 30 from the same tree in two ways. 30 has two children. In the first method the largest of the left subtree is sought. The left subtree of 30 is a single node (20); the largest is 20. 20 is written in place of 30 and the old node of 20 is deleted. The tree becomes 50(20(–, 40(35, –)), 70(60, 80)). Since the path 50 → 20 → 40 → 35 remains, the depth is 4. In the second method the smallest of the right subtree is sought: from 40 we go left to 35. 35 is written in place of 30 and the left of 40 becomes `NULL`. The tree is 50(35(20, 40), 70(60, 80)) and the depth drops to 3. In this example the second method gives a more balanced tree.

3. The smallest and the largest value. The smallest value is found by going left repeatedly from the root; it is at the first node with no left child. The largest value is found by going right repeatedly.

```
tree_ptr find_min(tree_ptr L) {
  if (L)
    while (L->left) L = L->left;
  return L;                /* NULL for an empty tree */
}
tree_ptr find_max(tree_ptr L) {
  if (L)
    while (L->right) L = L->right;
  return L;
}
```

Both procedures follow only one path and take *O(h)*. On the empty tree they return `NULL`. On the tree from the deletion examples, the smallest value came out as 10 and the largest as 73.

4. Checking a binary search tree. The permitted (lo, hi) range is passed to each node as parameters. When descending left, the upper bound becomes the value of the node; when descending right, the lower bound does.

```
#include <limits.h>
/* every value in L must satisfy lo < value < hi */
boolean in_range(tree_ptr L, long long lo, long long hi) {
  if (!L) return TRUE;
  if (L->data <= lo || L->data >= hi) return FALSE;
  return in_range(L->left, lo, L->data) &&
         in_range(L->right, L->data, hi);
}
boolean is_bst(tree_ptr L) {
  return in_range(L, (long long)INT_MIN - 1, (long long)INT_MAX + 1);
}
```

The bounds have type `long long`. This way the values `INT_MIN` and `INT_MAX` may also occur in the tree; the initial bounds lie just outside them. If `long` had been used, `INT_MAX + 1` would overflow on Windows, where `long` is 32 bits. Since each node is looked at once, the time is *O(n)* and the recursion depth is *O(h)*. The procedure found the tree from the deletion examples and its four deletion results valid. On a tree with root 15 and children 10 and 18, when 12 was placed to the left of 18, it returned `FALSE`. 12 is smaller than 18; a comparison with children alone would have accepted this tree as valid. But 12 is in the right subtree of 15 and cannot be smaller than 15.

5. Inserting 10, 20, 30, 25, 28 into an AVL tree. When 10 and 20 are inserted, the factor of 10 is −1. When 30 is inserted, the factor of 10 is −2 and the new value is on the right of the right (RR). 10 is rotated left and 20 becomes the root: 20(10, 30). 25 goes to the left of 30; the factors are in the range −1..+1. 28 goes to the right of 25. The factor of 30 becomes +2 and the new value is on the right of the left (LR). First 25 is rotated left: 28 takes the place of 25. Then 30 is rotated right: 28 becomes the parent of 25 and 30. The final tree is 20(10, 28(25, 30)). The driver program reported the same two rotations: `rot_left(10)` at 30, and `rot_left(25)` and `rot_right(30)` at 28.

6. The number of keys smaller than `x`. The simplest solution is to traverse the tree in inorder and count until `x` is reached; the time is *O(n)*. Using the order of the binary search tree, some subtrees can be skipped. If the root is not smaller than `x`, the right subtree is entirely larger and counting continues only on the left. If the root is smaller than `x`, the root itself and its whole left subtree are smaller. These are counted and counting continues on the right.

```
int node_count(tree_ptr L) {
  return L ? 1 + node_count(L->left) + node_count(L->right) : 0;
}
/* number of values smaller than x */
int count_less(tree_ptr L, int x) {
  if (!L) return 0;
  if (L->data >= x)                /* root and right are larger */
    return count_less(L->left, x);
  return 1 + node_count(L->left) /* root and left are smaller */
           + count_less(L->right, x);
}
```

In this version `node_count` still traverses the left subtrees from start to finish. So the worst case is still *O(n)*. If the size of its subtree is stored in a field at each node, `node_count` drops to *O(1)* and the counting is done along a single path, that is, in *O(h)*. This field must be updated at the nodes along the path at every insertion and deletion. In the example tree, 7 values smaller than 55 and 4 values smaller than 41 were found.

11.11 Summary

- In a **binary search tree** the values are unique; all values in the left subtree of every node are smaller than it, those in the right subtree are larger, and the subtrees are binary search trees too. Inorder traversal gives the values in increasing order.
- **Insertion** searches for the value from the root and puts it in place of the first `NULL`. The iterative version uses the pointer `q`, trailing one step behind, and the parameter `tree_ptr *L` for the empty tree. The insertion order determines the shape of the tree; sorted data builds a skewed tree.
- **Search** eliminates a subtree with every comparison and takes *O(h)*. The recursive version is tail recursion and returns the result of the recursive call unchanged. In an unsorted tree both sides must be examined and the search is *O(n)*.
- **Depth** is computed recursively with *MAX(left, right) + 1*; it makes *2n + 1* calls and is *Θ(n)*. Global variables and parameters carrying a hidden initial value are not used. If a recursive call is passed to a macro, the time becomes exponential.
- In **deletion**, leaves and nodes with one child are easy. For a node with two children, the value of the predecessor (or successor) is copied and its node is deleted; the link is updated in a single line with a pointer to a pointer. The deletion method can change the depth of the tree.
- **Balance.** On random data the average depth is *Θ(lg n)*, but on sorted data it rises to *n*. The **AVL tree** keeps the balance factor of every node in the range −1..+1; its height is at most about *1.44 lg(n + 2)*.

- In AVL insertion the imbalance is repaired with a single rotation in the LL and RR cases and a double rotation in the LR and RL cases. A rotation changes only two pointers and is *O(1)*; at most one rotation suffices in insertion.
- The **red-black tree** makes few rotations with a looser balance and is used in the map structures of libraries. The **B-tree** reduces disk accesses with its multi-key nodes and is used in databases.

11.12 Key terms

binary search tree (BST) key subtree leaf depth level preorder inorder postorder level-order skewed tree balanced tree self-balancing tree call by reference pointer to pointer global variable macro predecessor successor memory leak AVL tree balance factor rotation double rotation red-black tree B-tree splay tree recurrence relation amortized analysis

---

11.13 Test Yourself

1. The values 8, 3, 10, 1, 6, 14, 4, 7, 13 are inserted in this order into an empty binary search tree. What is the depth of the tree? What are the preorder and inorder sequences?

   Answer

   The tree is 8(3(1, 6(4, 7)), 10(–, 14(13, –))). The longest paths are 8 → 3 → 6 → 4 and 8 → 10 → 14 → 13; the depth is 4. The preorder is 8 3 1 6 4 7 10 14 13 and the inorder is 1 3 4 6 7 8 10 13 14.

2. In the same tree, 7 and 5 are searched for with `bst_search`. Which nodes are examined in each search, and what is the result?

   Answer

   For 7 the nodes 8, 3, 6, 7 are examined and the address of 7 is returned. For 5 the nodes 8, 3, 6, 4 are examined; since the right child of 4 is `NULL`, the search returns `NULL`.

3. The `bst_delete` procedure from the slides is applied to the same tree: first only 3 is deleted (in a

separate trial, only the root 8). How does the tree change in each case?

Answer

3 has two children. The largest of its left subtree is 1 (1 has no right child). 1 is written in place of 3 and the old node of 1 is deleted: 8(1(–, 6(4, 7)), 10(–, 14(13, –))). If the root 8 is deleted, the largest of the left subtree is found along the path 3 → 6 → 7: it is 7. 7 is written into the data field of the root and the right of 6 becomes `NULL`: 7(3(1, 6(4, –)), 10(–, 14(13, –))).

4. The preorder traversal of a binary search tree is 30 20 10 25 40 35 50. What is its postorder traversal?

Answer

The tree is 30(20(10, 25), 40(35, 50)). The postorder traversal is 10 25 20 35 50 40 30.

5. 1, 2, 3, 4, 5, 6, 7 are inserted in this order into an empty AVL tree. At which insertions is a rotation needed, and what is the final tree?

Answer

All the imbalances are RR cases and are repaired by a single left rotation. A left rotation is made at 1 when 3 is inserted, at 3 when 5 is inserted, at the root 2 when 6 is inserted, and at 5 when 7 is inserted; four rotations in all. The final tree is perfectly balanced: 4(2(1, 3), 6(5, 7)) and its height is 3.

6. What is the minimum number of nodes in an AVL tree of height 5?

Answer

From the recurrence $N(h) = N(h-1) + N(h-2) + 1$, $N(1) = 1$, $N(2) = 2$, we get $N(3) = 4$, $N(4) = 7$, $N(5) = 12$. At least 12 nodes are needed.

7. The one-line `depth` procedure is written with the `MAX` macro. On a right-extending chain of three nodes, how many times is the procedure called in total? How many times is the version with local variables called?

Answer

In the macro version $T(0) = 1$ and $T(n) = 2T(n-1) + 2$: $T(1) = 4$, $T(2) = 10$, $T(3) = 22$. The version with local variables makes $2n + 1 = 7$ calls. Both values were measured with a counter.

8. If the parameter of `bst_insert` is changed from `tree_ptr *L` to `tree_ptr L` and the last lines are corrected accordingly, what is the result of inserting 5 and 7 into an empty tree?

   Answer

   In the first insertion the new node is assigned only to the procedure's local copy of `L`. The caller's root stays `NULL`, and the node leaks since nothing points to it. The second insertion also finds the tree empty and the same thing happens. The tree stays empty and two nodes leak.

9. If the SWAP line in `proc` is moved in front of the two recursive calls, how does the tree with root 2, left child 1, right child 3, and left child 0 of 1 change?

   Answer

   First the children are exchanged at the root: 2(3, 1(0, −)). Then the left subtree (3) and the right subtree (1) are processed separately; at 1, 0 moves to the right. The final tree is 2(3, 1(−, 0)) and it is the exact mirror of the first tree. The inorder traversal is 3 2 1 0, that is, in decreasing order.

11.14 Further reading

- E. Horowitz, S. Sahni, and S. Anderson-Freed, *Fundamentals of Data Structures in C* (2nd ed.), Sections 5.7 and 10.2: the source of the binary search tree procedures and the AVL tree that these notes follow.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms* (3rd ed.), Chapters 12, 13, and 18: the binary search tree, the average depth of a randomly built tree, the red-black tree, and the B-tree, with proofs.
- M. A. Weiss, *Data Structures and Algorithm Analysis in C* (2nd ed.), Section 4.4: contains the AVL insertion example and the rotation code used in the notes; also introduces the splay tree.
- R. Sedgewick, *Algorithms in C*, Parts 1–4, Chapters 12–13: the average-case analysis of the binary search tree and a comparison of different kinds of balanced trees.

11.15 References

1. E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2nd ed., Sections 5.7 (binary search tree) and 10.2 (AVL tree).
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 3rd ed., MIT Press, Chapters 12 (binary search trees), 13 (red-black trees), 18 (B-trees).
3. M. A. Weiss, *Data Structures and Algorithm Analysis in C*, Section 4.4 (AVL trees).

4. R. Sedgewick, *Algorithms in C*, Parts 1–4, Chapters 12–13.

Threaded Binary Trees

In a linked binary tree, more than half of the pointer fields are empty. A threaded binary tree puts these empty fields to use. An empty left pointer points to the node that comes immediately before the node in inorder; an empty right pointer points to the node that comes immediately after. The tree can then be traversed from start to finish with a single pointer, with no stack and no recursion. These notes first prove how many empty pointers there are, then show how the threads are built, how they are told apart from real links, and what role the header node plays. Next come finding the successor, stack-free inorder and preorder traversal, the Morris traversal, which needs no flag fields, and the procedures `insert_right` and `insert_left`, which insert nodes without breaking the threads. For each procedure we give the C code, a line-by-line explanation, a code trace, and a complexity analysis.

LEARNING OUTCOMES

Counting null pointers
Prove that a binary tree with n nodes has n + 1 null pointers.

Building threads
Apply by hand, to a given tree, the rule that ties an empty left pointer to the inorder predecessor and an empty right pointer to the inorder successor.

Defining the structure completely
Build the C structure of a threaded tree, with the `left_thread` and `right_thread` flags and a header node.

Traversing without a stack
Traverse in inorder and preorder with `insucc` using O(1) extra memory; trace the Morris traversal step by step.

Inserting nodes
Merge the two cases of `insert_right` into a single procedure and derive `insert_left` from it.

Writing good code
Decide on a clearly defined procedure, a comment that states the intent, and the right place for the NULL check.

12.1 Empty pointers and the idea of a thread

In the linked representation, each node of a binary tree has three fields: a pointer to the left subtree, the data, and a pointer to the right subtree. In C the node is defined as a self-referential struct:

```
typedef struct node *tree_ptr;
struct node {
    tree_ptr left;   /* left subtree */
    char      data;
    tree_ptr right;   /* right subtree */
};
```

A pointer in a direction with no child is set to `NULL`. `NULL` is the **null pointer**, which points to no object. Both pointers of a leaf are null. A node with one child has one null pointer. The question for this chapter is simple: how many pointers in a tree are left null, and can these fields be put to any use?

12.1.1 n + 1 null pointers in n nodes

Whatever the shape of the tree, the number of null pointers depends only on the number of nodes. A counting argument shows this.

> **Theorem 12.1 (Number of null pointers)**
>
> Every binary tree with $n \geq 1$ nodes stored in the linked representation has exactly $n + 1$ pointer fields equal to `NULL`.

Proof. Each node has two pointer fields, so the tree has $2n$ fields in total. Every node other than the root receives exactly one pointer, from its parent. A node cannot receive pointers from two different fields, because every node has a single parent. No pointer from inside the tree reaches the root; the root is held by a variable outside the tree. Hence the number of non-null fields is $n - 1$, and the number of remaining fields is $2n - (n - 1) = n + 1$. ■

The proof never looks at whether the tree is balanced or skewed. Figure 12.1 illustrates this with three examples. The two trees on the left have 4 nodes and very different shapes; both have 5 null pointers. The 10-node tree on the right has 11 null pointers.

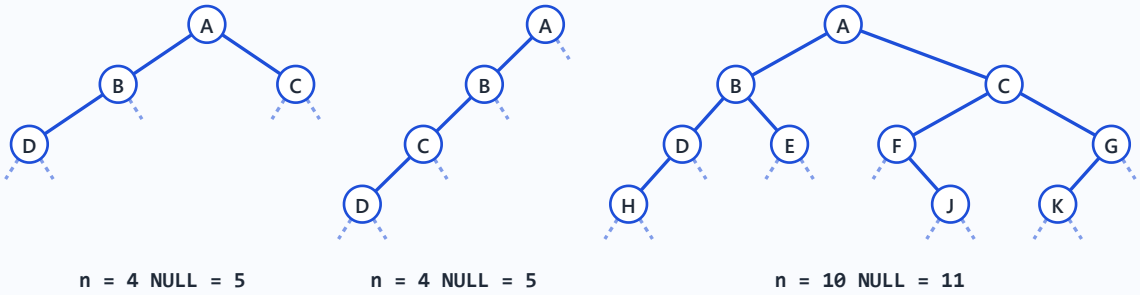


Figure 12.1: Three binary trees and their null pointers. The short dashed lines are `NULL` fields. The two 4-node trees have 5 null pointers and the 10-node tree has 11.

The ratio is striking. Of the $2n$ fields, $n + 1$, more than half, carry no information. On a 64-bit machine a pointer takes 8 bytes; in a tree of a million nodes, about 8 MB of memory goes to null pointers. These fields cannot simply be discarded, because it is not known in advance which node will have a child in which direction. But a meaningful value can be stored in them.

Example 12.1

The running example of these notes is a tree with nine nodes. A is the root; B and C are its children. D and E are the children of B; F and G are the children of C; H and I are the children of D. The tree has $2 \cdot 9 = 18$ pointer fields. Eight pointers arrive at the eight non-root nodes. The number of null fields is $18 - 8 = 10$: two for each of the leaves H, I, E, F, and G. This agrees with $n + 1 = 10$.

Common mistake

Counting the pointer to the root as a "non-null pointer" and getting n non-null fields. The variable that points to the root (for example `tree_ptr root`) is not a field of any node in the tree. The non-null fields inside the nodes number only $n - 1$.

12.1.2 The idea of Perlis and Thornton

In 1960 A. J. Perlis and C. Thornton proposed that null pointers could be used to point to other nodes of the tree. They called these pointers **threads**. A thread is not a real child of the node; it is a signpost that says which node to move to during a traversal.

The value of the idea shows in the extra memory a traversal needs. There are two familiar ways to traverse a binary tree in inorder. A recursive procedure opens one call for each node. The number of calls left open along one path is as large as the height h of the tree, so the call stack uses $O(h)$ memory. The non-recursive (iterative) procedure does the same job with an explicit stack: while going left it pushes the nodes it passes, and later returns to them. That stack also takes $O(h)$ space.

In a balanced tree h is about $log_2\ n$, and this cost is small. But in a **skewed tree** all nodes line up in one direction and $h = n$. In a left-skewed tree of a million nodes, a recursive traversal keeps a million calls open at the same time. On most systems the default call stack cannot hold that depth, and the program crashes.

In a threaded tree the situation is different. From each node, the next node in inorder, the **successor**, is reached either directly through a thread or by following a few real links. Because the way back is written inside the tree, nothing has to be remembered. The traversal is done with a single pointer variable, and the extra memory is *O(1)*. There is a price: for each pointer field we must keep one bit of information saying whether it is a real child or a thread.

Method	Time	Extra memory	What is remembered
Recursive	O(n)	O(h)	Open calls (the call stack)
Non-recursive, with a stack	O(n)	O(h)	The nodes passed while going left
Threaded	O(n)	O(1)	Nothing; the way back is written in the tree

Table 12.1: Three ways to traverse in inorder. h is the height of the tree; in a skewed tree $h = n$.

12.2 Building the threads

Which node an empty pointer should point to is not arbitrary. The rule is set by the inorder traversal sequence. So the notions of "previous" and "next" node in that sequence must first be made precise.

12.2.1 Inorder predecessor and successor

At each node, an inorder traversal visits the left subtree first, then the node itself, and the right subtree last. In the running example the sequence is `H D I B E A F C G`. In this sequence every node has a known neighbor on its left and one on its right.

Definition 12.1

The **inorder predecessor** of a node p is the node visited immediately before p in an inorder traversal. The **inorder successor** of p is the node visited immediately after p. The first node in the sequence has no predecessor, and the last node has no successor.

In the example sequence, the predecessor of E is B and its successor is A. H has no predecessor because it is the first node; G has no successor because it is the last. The rule for threads is written with these two notions.

Definition 12.2

A **threaded binary tree** is a binary tree whose null pointers have been filled according to the following two rules:

1. If `p->left_child == NULL`, then `p->left_child` points to the inorder predecessor of *p*.
2. If `p->right_child == NULL`, then `p->right_child` points to the inorder successor of *p*.

The pointers filled in this way are called **threads**.

Figure 12.2 shows the rules applied to the running example. The solid blue lines are real child links. The dashed red arrows are threads. The threads weave the inorder sequence of the tree into the tree itself: the red arrow leaving each leaf to the right goes to the next node in the sequence.

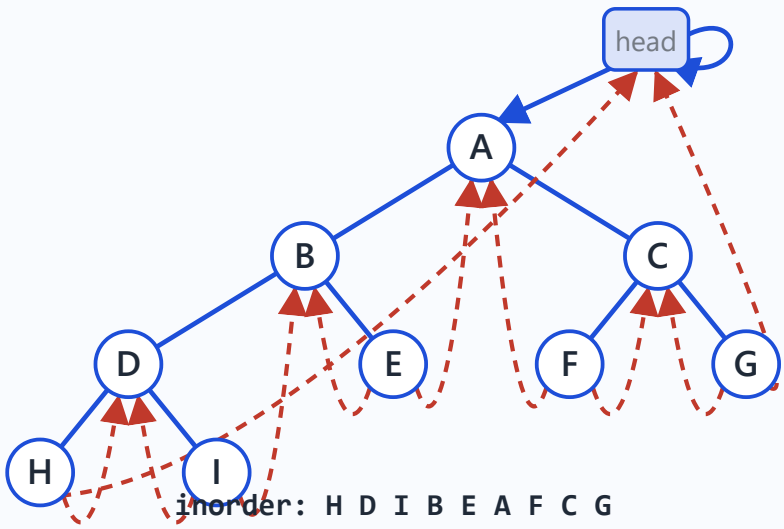


Figure 12.2: The running example with threads. The dashed red arrows are threads: a left thread goes to the predecessor and a right thread goes to the successor. The left thread of H and the right thread of G have no node to point to.

The threads have one property in common: all of them point upward to a node in the tree. The reason is this. The predecessor of a node with no left subtree is the nearest ancestor whose right subtree contains the node. Likewise, the successor of a node with no right subtree is the nearest ancestor whose left subtree contains the node. For example, I lies in the left subtree of B and has no right subtree; the successor of I is B. Going down is done with real links, and returning upward with threads. The information a stack has to remember is exactly these return paths.

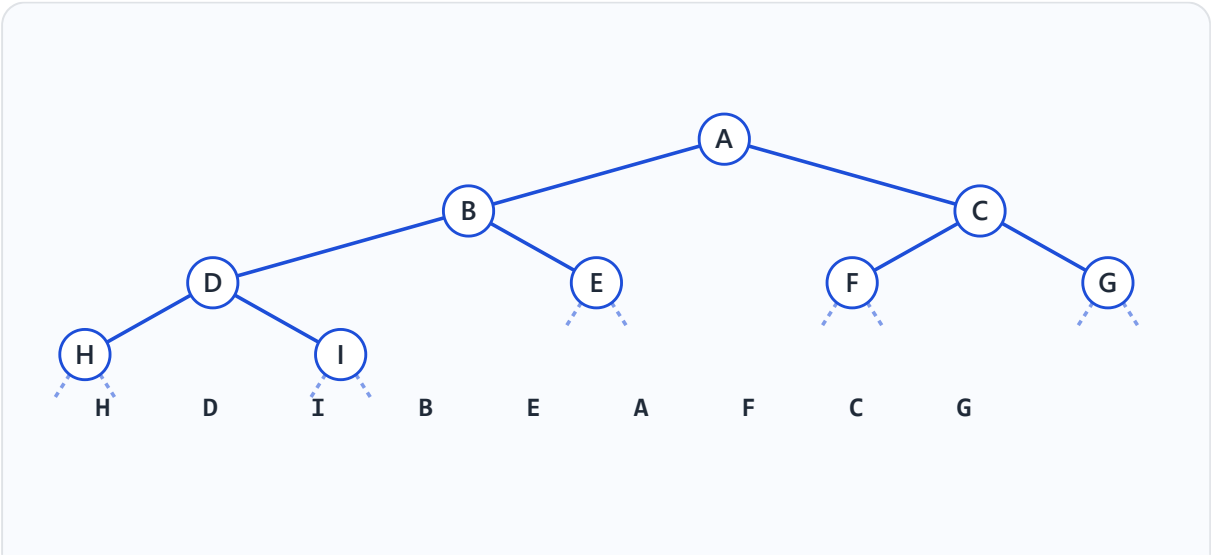
12.2.2 Building the threads one by one

To build the threads by hand, first write down the inorder sequence. Then, for each empty pointer, read off the node immediately to the left or to the right of the node in that sequence. The running example has ten empty fields (Example 12.1). Table 12.2 gives the target of each one.

Step	Empty field	Rule	Node pointed to
1	left of H	predecessor	none (head)
2	right of H	successor	D
3	left of I	predecessor	D
4	right of I	successor	B
5	left of E	predecessor	B
6	right of E	successor	A
7	left of F	predecessor	A
8	right of F	successor	C
9	left of G	predecessor	C
10	right of G	successor	none (head)

Table 12.2: Building the ten threads from the inorder sequence `H D I B E A F C G`. For the "head" rows there is no node to point to; these two threads will be tied to the header node.

Each row of the table corresponds to an adjacency in the sequence. The inorder sequence has eight pairs of neighbors (H–D, D–I, ..., C–G). Each pair of neighbors is tied by exactly one thread, even for pairs such as D and H that are also joined by a real link (the right thread of H points to D). The remaining two threads sit at the two ends of the sequence. Figure 12.3 shows the beginning and the end of the construction. At the top is the ordinary tree with its ten null pointers; at the bottom all ten threads are in place. The inorder sequence is also written under the lower figure, so one can check that each red arrow points to a neighbor in this sequence.



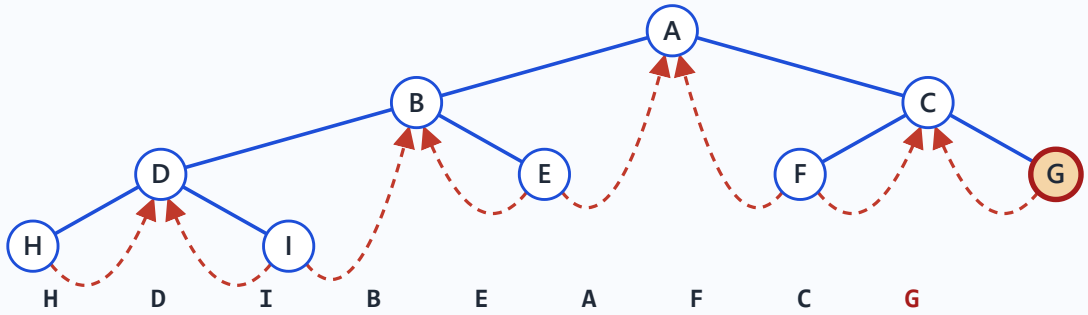


Figure 12.3: The first and last states of building threads. Top: the ordinary tree with 9 nodes and its 10 null pointers (short dashed lines). Bottom: all null pointers tied to a predecessor or successor; the left thread of H and the right thread of G are still dangling.

Example 12.2

Build the threads of the tree A(B(D, E), C(–, F)): A has children B and C, B has children D and E, and C has only a right child F. The inorder sequence is `D B E A C F`. The number of nodes is 6, and the number of empty fields is *6 + 1 = 7*.

- Left of D: no predecessor, so the head. Right of D: successor B.
- Left of E: predecessor B. Right of E: successor A.
- Left of C: predecessor A. C has a right subtree (F), so its right pointer is a real link.
- Left of F: predecessor C. Right of F: no successor, so the head.

There are 7 threads in all, 2 of which go to the header node. The left thread of C deserves attention: because the left of C is empty, the predecessor of C is A. A is the parent of C; this time the thread points to the parent.

12.2.3 Thread flags

On paper, threads are drawn in red and real links in black. To the compiler there is no difference between the two pointers: both are the address of a node. When a traversal procedure follows `p->left_child`, it cannot tell from the address alone whether it will descend to a real left child or return to a predecessor above. So next to each pointer we store information about its kind. Two extra fields are added to each node:

```
typedef struct threaded_tree *thread_ptr;
struct threaded_tree {
    short int    left_thread; /* TRUE: left_child is a thread */
    thread_ptr left_child;
    char         data;
    thread_ptr right_child;
    short int    right_thread; /* TRUE: right_child is a thread */
};
```

The structure adds two fields to the ordinary binary tree node. If `left_thread` is true (`TRUE`), then `left_child` is a thread, that is, it points to the predecessor; if it is false (`FALSE`), it points to a real left child. `right_thread` carries the same information for the right pointer. The constants `TRUE` and `FALSE` are defined with an enumeration type:

```
enum { FALSE, TRUE };      /* FALSE = 0, TRUE = 1 */
```

In an enumeration type the first name gets 0 and the second gets 1. So `FALSE` is 0 and `TRUE` is 1, and conditions such as `if (p->left_thread)` work directly. The `typedef` in the first line gives the type `struct threaded_tree *` the short name `thread_ptr`. This line comes before the definition of the structure; since the fields of the structure hold pointers to its own type, the name must be known first. The structure definition ends with a semicolon after the closing brace (line 8). If this semicolon is forgotten, the compiler usually reports the error on the next line, and the location of the mistake becomes confusing.

Don't the flags increase memory? They do, but the gain in time is much greater. The lecture states this decision openly: time complexity matters more than space complexity, and the two flags can in fact be stored in two bits. It is instructive to measure how much space is spent. The size of a structure is not just the sum of its fields. The compiler places each field on its own alignment boundary and inserts padding bytes between them. Table 12.3 gives the values measured on x86-64 with `sizeof` and `offsetof`.

Field	Offset (bytes)	Size	Padding after it
`left_thread`	0	2	6
`left_child`	8	8	0
`data`	16	1	7
`right_child`	24	8	0
`right_thread`	32	2	6

Table 12.3: Memory layout of `struct threaded_tree` (x86-64, pointer 8 bytes, `short` 2 bytes). The measured `sizeof` is 40 bytes.

Of the 40 bytes, 19 are padding. Pointers must be placed at addresses that are multiples of 8; the small fields in between each occupy a whole 8-byte slot by themselves. If the fields are ordered from large to small (first the two pointers, then the two `short`s, and `char` last), the same structure shrinks to 24 bytes. An ordinary node without flags is also 24 bytes. So if the field order is chosen well, the flags add no memory at all on this machine; they fit into the padding bytes that would otherwise be left empty behind the two pointers. Defining the flags as bit fields (`unsigned left_thread : 1`) does not change the result. The field order in the slides follows the node's drawing on paper: left flag, left pointer, data, right pointer, right flag. This order makes reading easier, but it is the most wasteful layout in memory.

Common mistake

Treating `left_child` or `right_child` as a real child without looking at the flag. In a threaded tree no pointer is `NULL`. A loop such as `while (p->left_child != NULL) p = p->left_child;`, written for an ordinary tree, never stops in a threaded tree. The left thread of the leftmost node goes to the head, and the left of the head goes back to the root; the loop runs forever. In a threaded tree every descent is checked with the flag.

12.2.4 The header node

In Table 12.2 two threads were left dangling. The left pointer of H should point to a predecessor, but H is the first node in the sequence. The right pointer of G should point to a successor, but G is the last node. Writing `NULL` into these two fields would break the "no pointer is null" arrangement and add special cases to the traversal procedures. The solution is the header node, familiar from linked lists.

Definition 12.3

The **header node** of a threaded tree is an extra node that carries no data and stands outside the tree. The root of the real tree is reached through `L->left_child`. The right pointer of the header always points to itself, and its `right_thread` field is `FALSE`. The two threads that have no node to point to, that is, the left thread of the first node and the right thread of the last node, point to the header.

In an empty tree there is only the header. In that case the left pointer of the header also points to itself, and `left_thread` is `TRUE`: what it points to is not a real root but a thread. When the first node is inserted, the left pointer is redirected to that node and `left_thread` is set to `FALSE`.

left_thread	left_child	data	right_child	right_thread
TRUE	→ head	–	→ head	FALSE

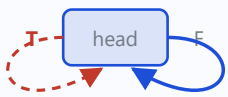
Table 12.4: The fields of the header, the only node of an empty threaded tree.

This arrangement is built by a C procedure. Two small helpers that do not appear in the slides are written. `new_node` allocates memory for a new node. `empty_threaded_tree` fills in the header as in Table 12.4:

```
thread_ptr new_node(char data) {
  thread_ptr d = malloc(sizeof *d);
  if (d == NULL) {              // out of memory
    fprintf(stderr, "out of memory\n");
    exit(EXIT_FAILURE);
  }
  d->data = data;
  d->left_child = d->right_child = NULL;   // insertion fills these in
  d->left_thread = d->right_thread = TRUE;
  return d;
}
thread_ptr empty_threaded_tree(void) {
  thread_ptr L = new_node('-');  // head: data unused
  L->left_child = L;  L->left_thread = TRUE;   // tree is empty
  L->right_child = L;  L->right_thread = FALSE;  // right is always itself
  return L;
}
```

`malloc` returns `NULL` if it cannot find the requested memory. Writing to `d->data` without checking the returned value means writing through a null pointer, which is undefined behavior. For this reason the procedure prints a message and ends the program when it gets `NULL`. The expression `sizeof *d` gives the size of the type that `d` points to. Because the type name is not repeated, the expression stays correct even if the structure changes. The pointers of the new node are set to `NULL`; when the node is inserted into a tree, `insert_right` or `insert_left` overwrites these values. For the same reason the initial values of the flags do not matter. In `empty_threaded_tree`, `'-'` is written into the data of the header; this value is never read.

The upper half of Figure 12.4 shows the empty tree and the lower half shows the running example with its header. The letters T and F in the figure are the thread flags on the two sides of each node. The left of the header goes to the root (A), and its left flag is F. The left thread of H and the right thread of G are now tied to the header. The right pointer of the header curls back onto itself.



left_child → itself (thread, TRUE) right_child → itself (FALSE)

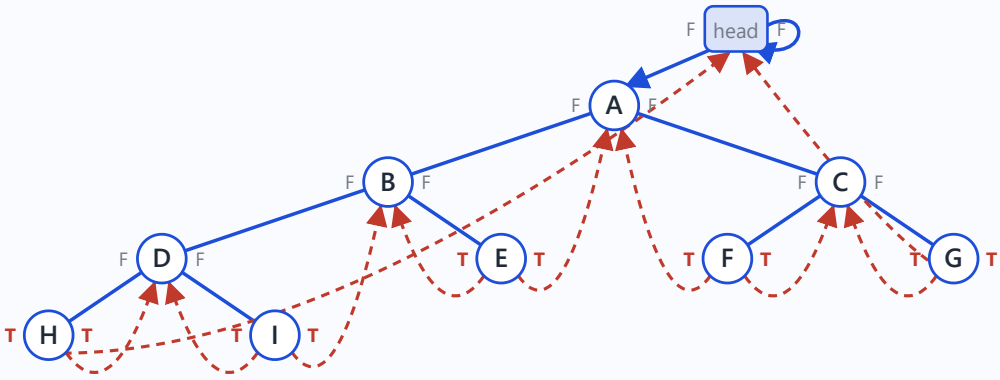


Figure 12.4: The header node. Top: the empty tree, where both pointers of the header point to itself. Bottom: the full tree, where the left of the header goes to the root and the two dangling threads are tied to the header. T and F are the thread flag of the node on that side.

With the header, the tree closes into a ring. In inorder, after the last node comes the header, and after the header the first node again. The traversal procedures use this ring: they start from the header and stop when they return to it. So special cases such as "is the tree empty", "which node is first", and "the last node has no successor" need no separate `if`.

12.2.5 Why inorder

The threads are built according to the inorder sequence. The three traversal orders of the same tree are side by side in Table 12.5.

Traversal	Rule	Order
inorder	left, root, right	`H D I B E A F C G`
preorder	root, left, right	`A B D H I E C F G`
postorder	left, right, root	`H I D E B F G C A`

Table 12.5: The three traversal orders of the running example.

The privilege of inorder comes from the consistency of the threads. In this order, the predecessor of a node with no left subtree is always an ancestor; so is the successor of a node with no right subtree. So an empty left field is exactly suited to point to the predecessor, and an empty right field exactly suited to point to the successor. Non-empty fields go downward, and the predecessor or successor is found at the far right or far left of that subtree. Together, the two rules make it possible to find the successor and the predecessor of every node by a short path. In preorder this symmetry is missing. For example, the preorder successor of D is H, and a real left link already goes from D to H; no information is left for a thread to add. For this reason the lecture stresses that the tree is built for inorder traversal. Preorder and postorder traversal can also be done with the same threads, but in a more roundabout way.

12.3 Inorder traversal

All traversal procedures of a threaded tree rest on a single question: which node is the inorder successor of a given node? If this question can be answered quickly, a traversal consists simply of starting at the header and moving from successor to successor.

12.3.1 Finding the successor

For the successor of a node *p* there are two cases.

- If `p->right_thread == TRUE`, then `p->right_child` is a thread. By the definition of a thread, this pointer already points to the successor of *p*. No search is needed.
- If `p->right_thread == FALSE`, then *p* has a real right subtree. After writing *p*, an inorder traversal enters this subtree and writes its leftmost node first. The successor is the leftmost node of the right subtree. To reach it, start from the right child and follow left links until `left_thread` is `TRUE`.

These two rules fit in a few lines in the procedure from the slides:

```
thread_ptr insucc(thread_ptr L) {
  // successor of the (non-NULL) node L
  thread_ptr p = L->right_child;
  if (!L->right_thread)        // is there a right subtree
    while (!p->left_thread)    // descend to its leftmost node
      p = p->left_child;
  return p;
}
```

The procedure works line by line as follows.

- **Lines 1–2.** The procedure takes a node pointer and returns the address of the successor. The comment states the precondition: `L` must not be `NULL`. In a threaded tree no pointer is `NULL`, so no such check is needed inside the procedure; satisfying the precondition is the caller's job.
- **Line 3.** `p` receives the right pointer of `L`. This value is both the result directly in the thread case and the starting point of the search in the right-subtree case. The first step common to the two cases is merged into a single assignment.
- **Line 4.** If `!L->right_thread` is true, the right pointer is a real link, that is, there is a right subtree. The comment writes the meaning of the question, not the flag: "is there a right subtree".
- **Lines 5–6.** In the right subtree we descend to the left. The loop condition looks at `p->left_thread`, not at `p->left_child`. The left pointer of the leftmost node is also a filled field (a thread), but its flag is `TRUE`.
- **Line 7.** In both cases `p` points to the successor, and it is returned.

The correctness of the rules follows from the definition of inorder traversal.

Theorem 12.2 (Successor rule)

In a threaded binary tree, if p has a right subtree, its successor is the leftmost node of that subtree. If it has no right subtree, its successor is the node that the thread `p->right_child` points to. `insucc` returns this node in both cases.

Proof. Immediately after writing p, an inorder traversal begins to traverse the right subtree of p. The traversal of a subtree always opens with its leftmost node: from the root we go down to the left, and the first node with no left child is written. So if there is a right subtree, the first node written after p is the leftmost node of that subtree. The loop of the procedure follows left links until the first node with no left child, that is, one whose `left_thread` is `TRUE`; this node is the leftmost node. If there is no right subtree, `p->right_child` is a thread and, by the second rule of Definition 12.2, points to the successor. ■

12.3.2 Code trace: head, A, and I

Let the procedure be called with three different nodes. Table 12.6 gives, for each call, the values `p` takes and the flags that decide, in order. The line numbers refer to the code above.

Call	Line	Event	p
`L` = head	3	`p = head->right_child`; the right of the head is itself	head
	4	`right_thread` is `FALSE`: the head is taken to have a real right subtree	head
	5–6	head-> `left_thread` is `FALSE`, go left	A
	5–6	A-> `left_thread` is `FALSE`, go left	B
	5–6	B-> `left_thread` is `FALSE`, go left	D
	5–6	D-> `left_thread` is `FALSE`, go left	H
	7	H-> `left_thread` is `TRUE`: the loop ends, H is returned	H
`L` = A	3	`p = A->right_child`	C
	4	`right_thread` is `FALSE`: A has a right subtree	C
	5–6	C-> `left_thread` is `FALSE`, go left	F
	7	F-> `left_thread` is `TRUE`: F is returned	F
`L` = I	3	`p = I->right_child` (a thread)	B
	4	`right_thread` is `TRUE`: the loop is not entered	B
	7	B is returned	B

Table 12.6: Trace of `insucc`. The tree is the running example with its header (Figure 12.4, bottom).

The first call gives a special result: the successor of the head is the first node in inorder (H). The right pointer of the head pointing to itself and its `right_thread` being `FALSE` are chosen for exactly this. The procedure thinks of the "right subtree" of the head as the head itself, starts descending to the left from there, and reaches the leftmost node of the real tree on the left of the head. So no separate "find the leftmost node" procedure is needed to start a traversal. Figure 12.5 shows the first and last moments of this call.

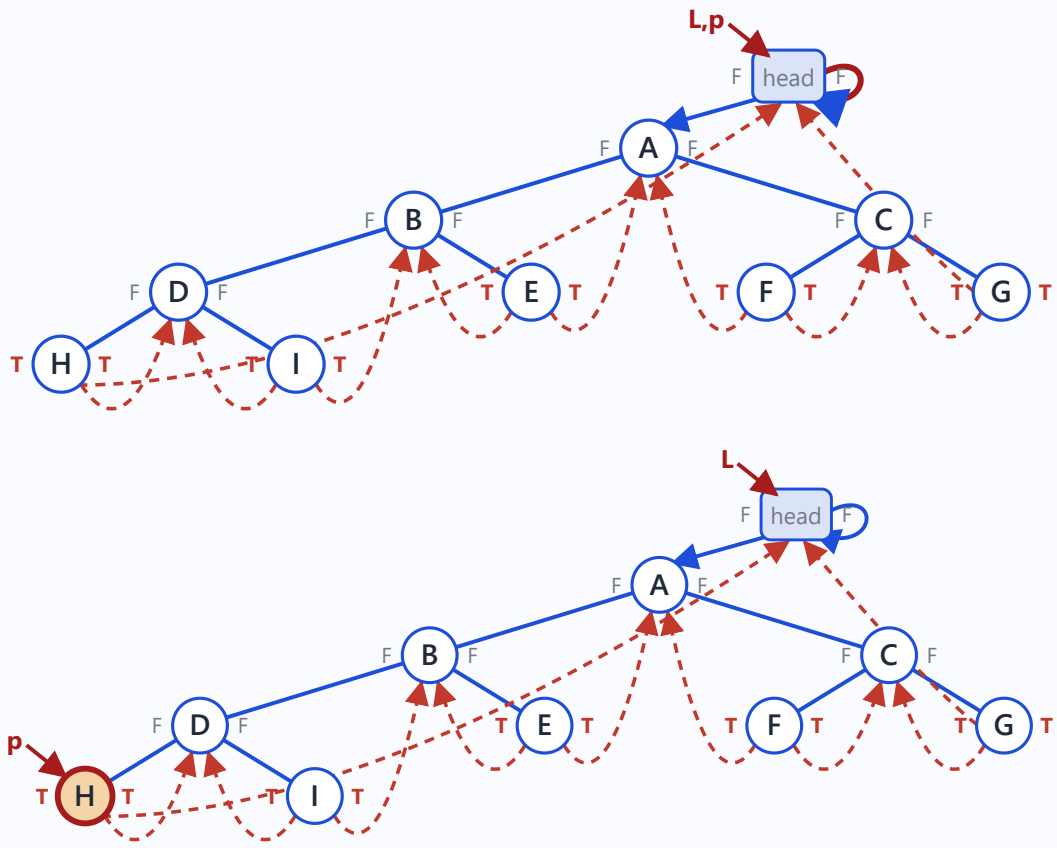


Figure 12.5: `insucc(head)`. Top, the start: `L` and `p` point to the head. Bottom: `p` has descended to H by following left links; since the left flag of H is T, the search stops.

In the call for A, the right subtree starts at C, and a single step to the left finds F. In the call for I there is no search at all; the right pointer is a thread and points directly to B. The successor of a leaf is always found in a single step like this.

Common mistake

Writing the loop as `while (!L->left_thread)` instead of `while (!p->left_thread)`. Since the condition looks at the same node on every iteration, the loop is either never entered or never left. The loop variable is `p`, and the condition must test its flag.

12.3.3 Traversal without a stack

Once the successor procedure is ready, inorder traversal is a short loop. We start at the header; on each iteration we move to the next node and write it. When we return to the header, the whole tree has been traversed.

```
void tinorder(thread_ptr L) {
  thread_ptr p = L;        // L: the header node
  for (;;) {
    p = insucc(p);
    if (p == L) break;       // back at the head?
    printf("%c ", p->data);
  }
}
```

- **Line 2.** `p` points to the header. The entire state of the traversal is in this single variable.
- **Line 3.** `for (;;)` is an unconditional loop; the exit is made by the `break` in the body. The condition cannot be put at the top of the loop, because the exit test has to be made after the successor is found.
- **Line 4.** `p` advances to its own successor.
- **Line 5.** If the successor is the header, the ring is complete. Since the right thread of the last node points to the header, this always happens.
- **Line 6.** The node is written.

In an empty tree the first call returns the header itself. `p = head->right_child` is the header. Because `right_thread` is `FALSE`, the loop is entered, but the `left_thread` of the header is `TRUE`, so it exits immediately. `p == L` is true, and the traversal ends without writing anything. Needing no separate check for the empty tree is the gain from the header node.

In the running example the loop runs ten times. Table 12.7 gives, for each iteration, the rule by which the successor was found and how many pointers were followed. The "Followed" column counts the read in line 3 and each descent to the left in the loop as one step each.

Iteration	`p` (before)	How it was found	`p` (after)	Followed	Output
1	head	leftmost of the right subtree (A, B, D, H)	H	5	`H`
2	H	directly through a thread	D	1	`D`
3	D	leftmost of the right subtree (I)	I	1	`I`
4	I	directly through a thread	B	1	`B`
5	B	leftmost of the right subtree (E)	E	1	`E`
6	E	directly through a thread	A	1	`A`
7	A	leftmost of the right subtree (C, F)	F	2	`F`
8	F	directly through a thread	C	1	`C`

Iteration	`p` (before)	How it was found	`p` (after)	Followed	Output
9	C	leftmost of the right subtree (G)	G	1	`G`
10	G	directly through a thread	head	1	(`break`)

Table 12.7: Trace of `tinorder`. The output is `H D I B E A F C G`; 15 pointers are followed in total.

The two rules in the table follow one another in turn. After a leaf we go up through a thread; after an internal node we enter the right subtree and descend to the far left. No node is written twice in any iteration, and a stack is needed nowhere.

12.3.4 The cost of the traversal

A single call of `insucc` can take a long time. The first call, made with the header, descends from the root to the leftmost node; this path is as long as the height of the tree. The worst cost of a single call is therefore $O(h)$. In a left-skewed tree of 1000 nodes this first call follows 1001 pointers. But the total cost of the whole traversal is far smaller than the product of the number of calls and the worst cost.

Theorem 12.3 (Cost of a threaded traversal)

Traversing a threaded tree of n nodes with `tinorder` follows at most $2n + 1$ pointers. The time is $\Theta(n)$ and the extra memory is $O(1)$.

Proof. The loop calls `insucc` $n + 1$ times: once for each of the n nodes and once for the return to the header. Each call reads exactly one pointer in line 3, which adds up to $n + 1$ steps. The descents to the left in the loop go only over real left links. A left link is followed only when entering, from the left, the subtree that contains the node owning the link, and the traversal enters each subtree once. So each real left link is followed at most once. The number of real left links, counting the link from the header to the root, is at most n. The total is $(n + 1) + n = 2n + 1$. Each step takes constant time, so the time is $O(n)$. Since every node is written, the time is also $\Omega(n)$. The procedure uses only the variable `p`; the extra memory is $O(1)$. ■

In the running example, the nodes with a left child are A, B, C, and D. Together with the link from the header to the root, five descents to the left are made. The total $10 + 5 = 15$ steps equals the sum of the last column of Table 12.7. The extreme cases were also checked by counting. In a left-skewed tree of 1000 nodes every node has a left child and the bound is reached exactly: 2001 steps. In a right-skewed tree of 1000 nodes only one descent to the left is made, from the header to the root: $1001 + 1 = 1002$ steps.

Dividing the total of $2n + 1$ steps among $n + 1$ calls leaves fewer than two steps per call. Although there are expensive calls when looked at one by one, they are few in a traversal; an expensive descent makes the following calls cheap. This kind of accounting is called **amortized analysis**. The amortized cost of finding a successor within a traversal is $O(1)$.

Threads bring one more gain. Once the address of any node is at hand, its successor can be reached without starting the traversal from the beginning. In an ordinary tree, finding the successor of a node requires either a parent pointer or a fresh descent from the root. A threaded tree answers this question starting from the node itself.

The value of such an improvement should not be underestimated. The lecture gives an example: a problem is known to be solved in $O(n^2)$ time, and someone works out an algorithm that brings it down to $O(n^{1.89})$. This small difference in the exponent is a big achievement, and that algorithm goes into the literature. Threaded traversal improves not time but memory: it brings $O(h)$ down to $O(1)$. In skewed trees that means going from $O(n)$ to $O(1)$.

12.3.5 Preorder traversal

The threads are built for inorder, but the same threads can also be used for preorder traversal. For this we write a procedure that finds the preorder successor.

```
thread_ptr presucc(thread_ptr p) {
  if (!p->left_thread)        // the left child, if any
    return p->left_child;
  while (p->right_thread)      // climb the right threads
    p = p->right_child;
  return p->right_child;        // first real right child
}
```

In a preorder traversal, after a node is written, its left subtree is entered if there is one; the first node of the left subtree is its root. Lines 2–3 handle this case: if the left flag is `FALSE`, the successor is the left child. If there is no left child, the successor is the root of the first not-yet-traversed right subtree found above, starting from *p*. Lines 4–5 look for this node. As long as the right pointer of *p* is a thread, the thread is followed. The right thread goes to the inorder successor, that is, to the nearest ancestor that carries *p* in its left subtree. The left subtree of that ancestor is finished, and next comes the right subtree of the ancestor. If that ancestor also has no right subtree (its flag is again `TRUE`), one more thread is followed. We stop at the first node with a real right link, and line 6 returns its right child.

The traversal loop using this procedure has the same form as the inorder traversal:

```
void tpreorder(thread_ptr L) {
  thread_ptr p = L;        // L: the header node
  for (;;) {
    p = presucc(p);
    if (p == L) break;       // back at the head?
    printf("%c ", p->data);
  }
}
```

Starting from the header, the left flag of the header is `FALSE`, so the successor is the root (A). After the last node, the threads climb up to the header; the right flag of the header is `FALSE` and its right pointer is itself. The procedure returns the header and the traversal ends. Table 12.8 gives all the steps.

`p`	Left flag	Right threads followed	Successor
head	F	–	A (left child)
A	F	–	B (left child)
B	F	–	D (left child)
D	F	–	H (left child)
H	T	H → D	I (right child of D)
I	T	I → B	E (right child of B)
E	T	E → A	C (right child of A)
C	F	–	F (left child)
F	T	F → C	G (right child of C)
G	T	G → head	head (the right of the head is itself): done

Table 12.8: Traversal with `presucc`. The output is `A B D H I E C F G`.

Finding the preorder successor of a leaf may require following several threads. For example, after I we climb to B through one thread and find the right child of B, E. In a deeper tree, starting from a leaf, several ancestors in a row may have no right subtree; in that case the chain of threads gets longer. Even so, each right thread is followed at most once in the whole traversal, so the total time stays *O(n)* and the extra memory *O(1)*.

Postorder traversal cannot be written as easily. In postorder, a node is usually followed by its parent. If the node is the right child of its parent, its successor is the parent; if it is the left child, its successor is the first postorder node in the parent's right subtree. In both cases the parent must be known. Threads point not to the parent but to the inorder predecessor and successor, and these two do not always coincide with the parent. The right thread of H points to its parent D, but the right thread of I points not to its parent D but to B. Postorder traversal with threads is possible but roundabout. Adding a parent pointer to each node makes the job easier.

12.3.6 The Morris traversal

A threaded tree pays two costs: flag fields in every node, and the extra complexity of the insertion procedures. In 1979 J. M. Morris proposed a method that needs no flag fields. The **Morris traversal** builds the threads temporarily during the traversal in an ordinary binary tree and removes them when it is done. At the end of the traversal the tree returns to its original state. The procedure works with the ordinary `struct node` node given at the beginning of the chapter:

```c
void morris_inorder(tree_ptr L) {
  while (L) {
    if (!L->left) {             /* no left: print, go right */
      printf("%c", L->data);  L = L->right;
    } else {
      tree_ptr p = L->left;    /* inorder predecessor */
      while (p->right && p->right != L)
        p = p->right;
      if (!p->right) {           /* set a temporary thread */
        p->right = L;  L = L->left;
      } else {                      /* remove the thread */
        p->right = NULL;
        printf("%c", L->data);  L = L->right;
      }
    }
  }
}
```

The idea of the procedure is this. When we arrive at a node L that has a left subtree, we prepare the way back before going left. The inorder predecessor of L is the rightmost node of its left subtree. The right pointer of this node is null, because the rightmost node cannot have a right child. We write the address of L into this empty field: a temporary thread. After the left subtree has been traversed and the predecessor is reached, following this thread leads back to L. When we reach L a second time, the predecessor is searched for again; this time the right pointer of the predecessor is seen to point to L. This is the sign that the left subtree is finished. The thread is removed, L is written, and we move right.

- **Line 2.** The loop runs until `L` becomes `NULL`. The right pointer of the last node really is `NULL`; the loop ends there.
- **Lines 3–4.** If there is no left subtree, the node is written and we move right. The right pointer may be a real child or a temporary thread set earlier. The procedure does not distinguish the two; in both cases it goes to the right node.
- **Lines 6–8.** The predecessor is searched for: start from the left child and go right as long as possible. The second half of the condition (`p->right != L`) is important. If the thread has been set earlier, the right pointer of the rightmost node points to `L`; without this test the search would follow the thread to `L` and from there turn left again.
- **Lines 9–10.** If the right of the predecessor is null, this is the first visit: the thread is set and we descend to the left.
- **Lines 11–13.** If the right of the predecessor points to `L`, this is the second visit: the thread is removed, `L` is written, and we move right.

Let us trace the procedure on the tree A(B(D, E), C): the five-node tree in which A has children B and C, and B has children D and E. Table 12.9 shows each iteration and Figure 12.6 shows four moments.

Iteration	`L`	Predecessor `p`	Event	Temporary threads	Output
1	A	E (right from B)	the right of E is empty: set the thread, go left	E→A	
2	B	D	the right of D is empty: set the thread, go left	E→A, D→B	
3	D	–	no left child: write, go right (through the thread to B)	E→A, D→B	`D`
4	B	D	the right of D is B: remove the thread, write, go right	E→A	`DB`
5	E	–	no left child: write, go right (through the thread to A)	E→A	`DBE`
6	A	E (right from B)	the right of E is A: remove the thread, write, go right	–	`DBEA`
7	C	–	no left child: write, go right (`NULL`)	–	`DBEAC`

Table 12.9: Trace of `morris_inorder`, tree A(B(D, E), C). The output is `DBEAC`.

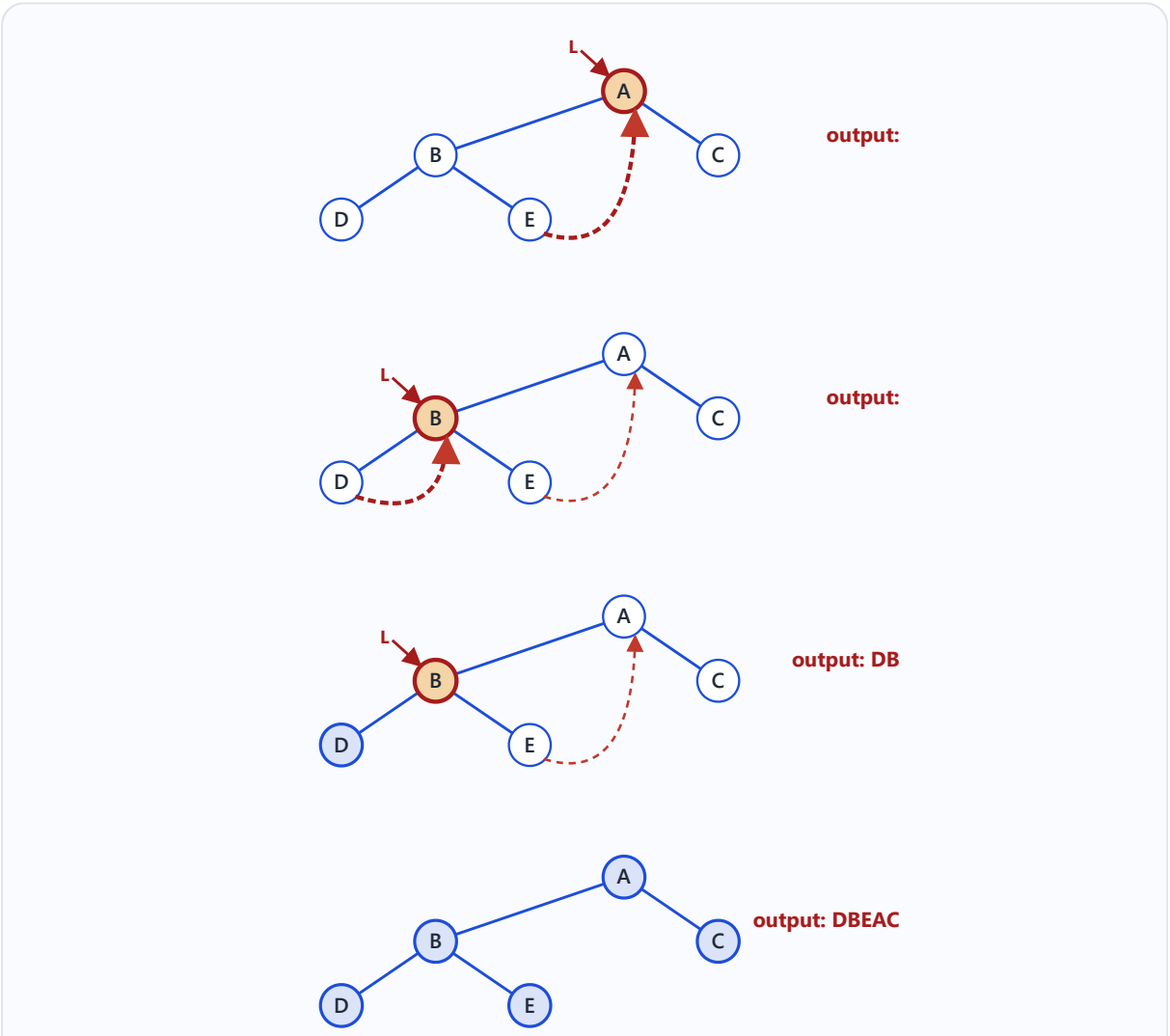


Figure 12.6: Four moments of the Morris traversal. In the first, the thread E → A has been set and we have gone left. In the second, the thread D → B has been set as well. In the third, we have returned from D to B, the

thread D → B has been removed, and B has been written. In the fourth, the traversal is finished; the tree is in its original state.

The table shows that every temporary thread is seen exactly twice: when it is set and when it is removed. Every node with a left child gives rise to one thread; in this tree there are two (for A and B). When the traversal ends, the right pointers of D and E are `NULL` again. The result was checked by comparing it with the output of a recursive inorder traversal of the same tree:

```
void inorder(tree_ptr p) {
  if (p) {
    inorder(p->left);
    printf("%c", p->data);
    inorder(p->right);
  }
}
```

The time of the Morris traversal is also *O(n)*. For each node with a left child the predecessor is searched for twice. Each search walks a path that starts at the left child and goes right. These paths for different nodes share no edge; an edge lies on the predecessor path of at most one node. The total length of the paths is therefore at most $n - 1$, and walking them twice bounds the total by $2(n - 1)$. To this is added the constant work done at each node. Measurement confirms this: the five-node example makes 13 pointer moves in all, and the complete seven-node tree makes 18.

The method has a limit. While the traversal is in progress, the tree is temporarily modified: some `NULL` fields point to other nodes. If another thread reads the tree during this time, it meets cycles and gets a wrong result. If the traversal is cut short by an error or an early exit, the tree is left corrupted. The Morris traversal is therefore suitable for procedures that have the tree to themselves and carry the traversal through to the end.

12.4 Inserting a node

Traversal in a threaded tree is easy; the price is paid at insertion. Inserting a node into a binary search tree is a single pointer assignment: the new node is written into an empty child field. In a threaded tree there are no empty fields. When a new node is inserted, both its own threads must point to the right places and the threads around it must be adjusted to the new order. Insertion is therefore not an easy procedure on its own; it is a little intricate.

12.4.1 Why insertion is intricate

A new node is inserted either to the right or to the left of a parent. The pointer assignments for left and right insertion are mirror images of each other, but they are not literally identical; so two separate procedures are written: `insert_right` and `insert_left`. Let us deal with `insert_right`

first. There are two cases in inserting on the right:

1. The parent has no right subtree. The new node is placed to the right of the parent as a leaf.
2. The parent has a right subtree. The new node goes in between the parent and this subtree; the old subtree becomes the right subtree of the new node.

The second case never arises in inserting into an ordinary binary search tree; there the new node is always a leaf. In a threaded tree both cases are useful: the second is a way to put a node into the middle of the tree.

12.4.2 Case 1: inserting under a parent with no right subtree

Let the tree be A(–, B(C, –)): A is on the left of the head; A has no left child and has right child B; B has left child C and no right child. The inorder sequence is `A C B`. The new node D will be inserted to the right of B. The right pointer of B is a thread, and since B is the last node in the sequence, it points to the head. The insertion is done with six assignments:

```
child->right_child  = parent->right_child;
child->right_thread = TRUE;  // parent had no right child
child->left_child   = parent;
child->left_thread  = TRUE;
parent->right_child   = child;
parent->right_thread = FALSE;  // was TRUE
```

The reason for each assignment can be read off the inorder sequence. After the insertion the sequence will be `A C B D`. D comes immediately after B; the predecessor of D is B, and its successor is the old successor of B. Table 12.10 follows the assignments in order, and Figure 12.7 shows the first and last states.

Line	Assignment	Result	Reason
1	`child->right_child = parent->right_child`	D → head	The successor of the parent is now the successor of the child. It does not matter whether this successor is the head.
2	`child->right_thread = TRUE`	red arrow	The parent had no right subtree; the child, being a leaf, has none either.
3	`child->left_child = parent`	D → B	The predecessor of the child is the parent.
4	`child->left_thread = TRUE`	red arrow	This is a thread as well.
5	`parent->right_child = child`	B → D	A real link from the parent to the child.
6	`parent->right_thread = FALSE`	black link	The right of the parent is no longer a thread but a real child.

Table 12.10: Trace of Case 1: `parent` = B, `child` = D.

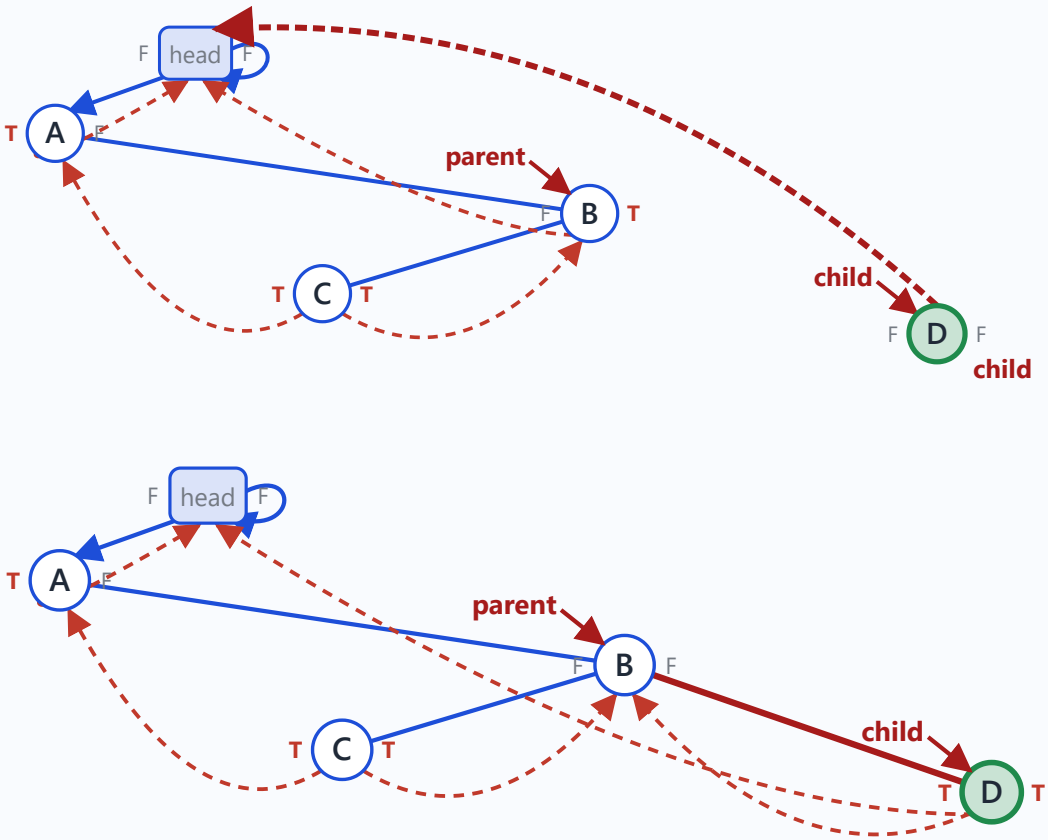


Figure 12.7: Case 1. Top, after the first assignment: D is not yet attached to the tree; its right pointer points to the head, which is the successor of B. Bottom, after the six assignments: D is the real right child of B; the left thread of D points to B and its right thread points to the head.

The order of the assignments is not arbitrary. Line 1 reads the right pointer of the parent, and line 5 overwrites it. If the places of these two lines are exchanged, `child->right_child` receives the child's own address instead of the old successor of the parent, and the successor information is lost. The general rule is this: if the old value of a field is needed, use that value first and only then change the field. All the assignments take constant time; the cost of Case 1 is *O(1)*.

12.4.3 Case 2: the parent has a right subtree

Now let the tree be A(–, B(C, D(E, F))). The right child of B is D, and the children of D are E and F. The inorder sequence is `A C B E D F`. The new node X will be made the right child of B. What we want is that the subtree of D must not be lost and must become the right subtree of X. The red pointers must again point to the right places. After the insertion the sequence will be `A C B X E D F`.

Two requirements are read from this sequence. The predecessor of X is B; this is as in Case 1. But the node that comes after X is E, the leftmost node of the subtree of D. The left thread of E used to point to B, because the old predecessor of E was B. After the insertion the predecessor of E becomes X. So the child must be made the predecessor of the old successor of the parent.

What happens if the six assignments of Case 1 are applied as they are to this tree? The result of trying it is this: the traversal writes `A C B X D F`; E is lost. The cause is line 2. `child->right_thread = TRUE` treats the right pointer of X (which points to D) as a thread. After X, the traversal jumps directly to D by the thread rule and never descends into the left subtree of D. The left thread of E also still points to B. The tree is corrupted, and no compiler warning announces it.

The fix has two parts:

- Line 2 is written as `child->right_thread = parent->right_thread`. In Case 1 the flag of the parent is `TRUE`, and the thread status passes to the child; in Case 2 it is `FALSE`, and the real-link status is preserved. The subtree of the parent has now become the subtree of the child.
- In Case 2, the left thread of the old successor of the parent, that is, of the successor of the child in the new arrangement, is redirected to the child.

With these two changes a single procedure that handles both cases can be written.

12.4.4 One procedure: `insert_right`

```
void insert_right(thread_ptr parent,
                  thread_ptr child) {
  // make child the right child of parent (threaded)
  thread_ptr p;
  child->right_child   = parent->right_child;
  child->right_thread = parent->right_thread;
  child->left_child   = parent;
  child->left_thread = TRUE;
  parent->right_child    = child;
  parent->right_thread  = FALSE;
  if (!child->right_thread) { // did parent have a subtree?
    p = insucc(child);
    p->left_child = child;     // successor's predecessor is child
  }
}
```

The procedure reads line by line as follows. The line numbers refer to the code above.

- **Lines 1–2, a clear definition.** The name of the procedure says what it does; the incoming parameters (`parent`, `child`) are clear; there is no return value (`void`). A clearly defined procedure can be used without reading its body.
- **Lines 5–6.** The right pointer and right flag of the child are taken from the parent. In Case 1 the thread status (`TRUE`) is kept, and in Case 2 the real-link status (`FALSE`). The common work is written in a common form.
- **Lines 7–8.** The predecessor of the child is the parent: the left pointer goes to the parent and the left flag becomes `TRUE`.
- **Line 9.** A real link from the parent to the child.

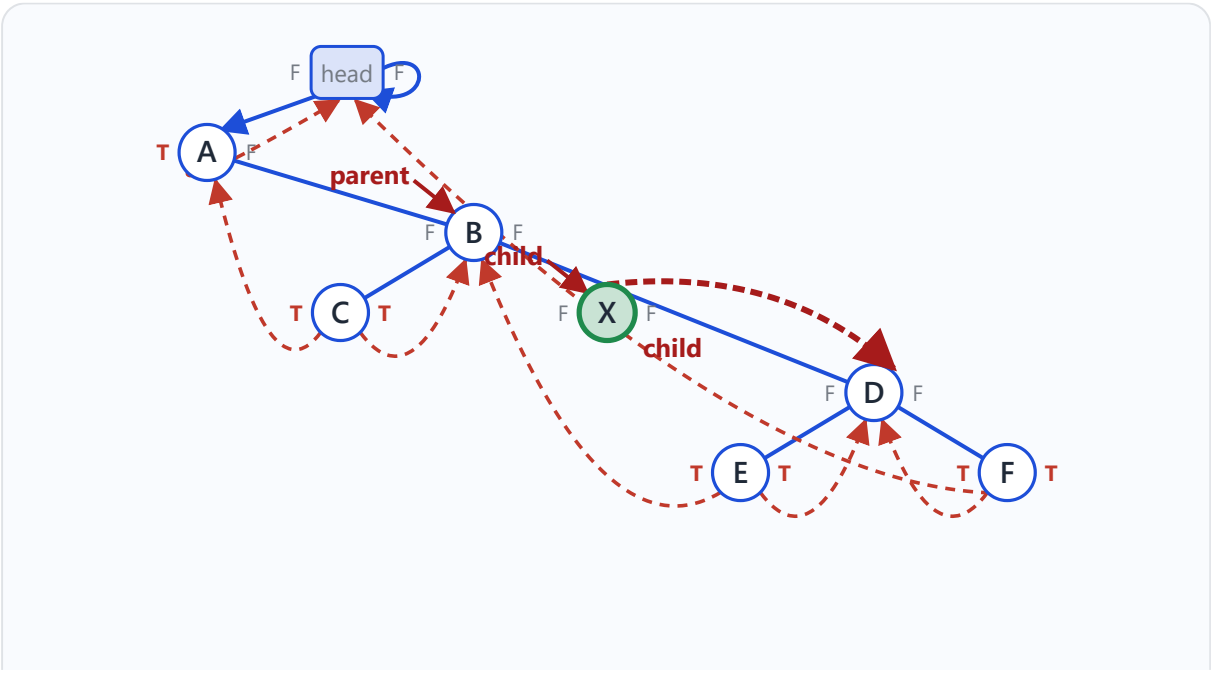
- **Line 10.** The right flag of the parent is set to `FALSE`. In Case 2 the flag is already `FALSE`, so this assignment is unnecessary but harmless; in Case 1 it is needed. Writing it in both cases, instead of separating them with an `if`, simplifies the code.
- **Lines 11–13.** If the right flag of the child is `FALSE`, the child has a real right subtree; this is Case 2. The successor of the child is found and its left thread is redirected to the child. In Case 1 only one test is made and the procedure exits.

The condition in line 11 tests a flag, but the question it asks is this: did the parent have a right subtree? After lines 5–6, the old right information of the parent has been moved to the child, so this question is answered by looking at the flag of the child. The comment writes this question, not the flag.

Table 12.11 traces the call `insert_right(B, X)` line by line on the Case 2 tree.

Line	Event	Changed field
5	The old right subtree of B goes to the right of X	X-> `right_child` = D
6	The flag of B was `FALSE`; the real-link status is kept	X-> `right_thread` = `FALSE`
7	The predecessor of X is B	X-> `left_child` = B
8	red arrow	X-> `left_thread` = `TRUE`
9	A real link from B to X	B-> `right_child` = X
10	it was already `FALSE`; harmless	B-> `right_thread` = `FALSE`
11	`!X->right_thread` is true: X has a right subtree	–
12	The successor of X: descend left from D and stop at E	`p` = E
13	The left thread of E points to X instead of B	E-> `left_child` = X

Table 12.11: Trace of `insert_right(B, X)`, tree A(–, B(C, D(E, F))). The resulting sequence is `A C B X E D F`.



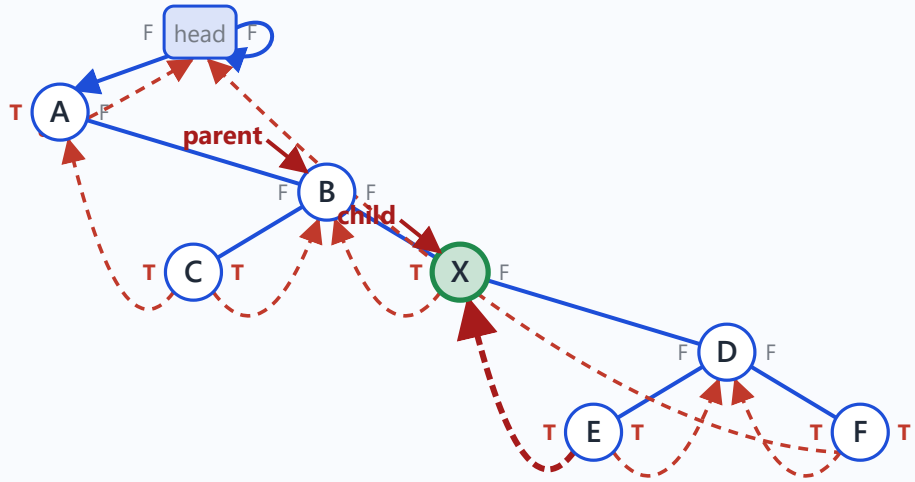


Figure 12.8: `insert_right(B, X)`, Case 2. Top, after the first assignment: the right of X points to D, X is not yet attached to the tree, and the left thread of E points to B. Bottom, after the procedure finishes: X is between B and D; the left thread of E points to X.

After line 10 the tree is almost correct; X is in place, and even the traversal writes `A C B X E D F`, because the traversal does not use left threads. But the left thread of E still points to B. A procedure that traverses backwards in inorder or looks for a predecessor exposes this bug at once. The lecture warns at this point: one should not decide right away that the code works correctly; there is still an `if` block to be executed.

The same procedure was also tried on the Case 1 tree. `insert_right(B, D)` on the tree A(–, B(C, –)) gives the same result as the six-assignment code: the sequence `A C B D`, the left thread of D pointing to B, and its right thread pointing to the head. Because the flag of the parent is `TRUE`, the condition in line 11 is false and the `if` block does not run.

Cost. Lines 5–10 take constant time. The successor search in line 12 descends to the left in the right subtree of the child; this path is at most as long as the height of the tree. The time of `insert_right` is *O(h)* in the worst case and *O(1)* in Case 1. The extra memory is *O(1)*.

12.4.5 Inserting on the left

`insert_left` makes the node pointed to by child the left child of the parent. The procedure is the mirror image of `insert_right`: left and right, and predecessor and successor, are exchanged. The old predecessor of the parent becomes the predecessor of the child. If the parent has a left subtree, the right thread of the rightmost node of that subtree, that is, of the new predecessor of the child, is redirected to the child. For this we need a procedure that finds the inorder predecessor. The predecessor rule is the mirror image of the successor rule: the rightmost node of the left subtree if there is one, otherwise the node that the left thread points to.

```
thread_ptr inpred(thread_ptr L) {
  thread_ptr p = L->left_child;
  if (!L->left_thread)
    while (!p->right_thread) p = p->right_child;
  return p;
}
void insert_left(thread_ptr parent, thread_ptr child) {
  child->left_child   = parent->left_child;
  child->left_thread = parent->left_thread;
  child->right_child   = parent;
  child->right_thread = TRUE;      // successor is parent
  parent->left_child    = child;
  parent->left_thread  = FALSE;
  if (!child->left_thread) {      // parent had a left subtree
    thread_ptr p = inpred(child);
    p->right_child = child;         // predecessor's successor is child
  }
}
```

`insert_right`	`insert_left`
`child->right_child = parent->right_child`	`child->left_child = parent->left_child`
`child->right_thread = parent->right_thread`	`child->left_thread = parent->left_thread`
`child->left_child = parent` (the parent is its predecessor)	`child->right_child = parent` (the parent is its successor)
`parent->right_thread = FALSE`	`parent->left_thread = FALSE`
left thread of the successor to the child	right thread of the predecessor to the child
`insucc` : right, then left and left	`inpred` : left, then right and right

Table 12.12: The mirror relation between `insert_right` and `insert_left` .

Example 12.3

Let the tree be A(B(D, E), C); the inorder sequence is `D B E A C` . Let `insert_left(A, X)` be called. A has a left subtree (B); this is the second case. The lines (8–16) of the second procedure in the code do the following in order:

1. The left of X takes the old left of A: X-> `left_child` = B; the flag is carried over as `FALSE` .
2. The right of X points to A and is `TRUE` : the successor of X is A.
3. A-> `left_child` = X, A-> `left_thread` = `FALSE` .
4. The left flag of X is `FALSE` ; the predecessor of X is searched for. `p` = X-> `left_child` = B. The right flag of B is `FALSE` , go right: `p` = E. The right flag of E is `TRUE` , stop.
5. E-> `right_child` = X: the right thread of E points to X instead of A.

The resulting sequence is `D B E X A C` . X has been inserted between the B subtree and A.

The procedure `inpred` can be used for another job as well: starting from the header and moving from predecessor to predecessor until we return to the header traverses the tree backwards in inorder. This is taken up in the exercises. The predecessor of the header is the last node in inorder: the left flag of the header is `FALSE`, its left pointer is the root, and going right and right from the root reaches the rightmost node.

12.4.6 Freeing the tree

In a C program, every block obtained with `malloc` must be returned with `free` when it is no longer needed. Blocks that are not returned stay in memory until the program ends. This is called a **memory leak**. In a long-running program a leak slowly exhausts memory. A threaded tree is freed by a traversal too:

```
void threaded_tree_free(thread_ptr L) {
  thread_ptr p = insucc(L), next;
  while (p != L) {
    next = insucc(p);  // find the successor first
    free(p);                          // then free the node
    p = next;
  }
  free(L);                            // the head last
}
```

The order matters. Reading the fields of a node after it has been returned with `free` means reading freed memory, which is undefined behavior. So the successor is found first and written into `next`, and only then is the node freed. The successor procedure reads only the fields of `p` and of the nodes in its right subtree. These nodes have not been freed yet, because they come after `p` in inorder. The header is freed last; the stopping condition of the traversal looks at its address.

All the code in these notes was tested with a wrapper that counts calls to `malloc` and `free`. The code from the slides and the additional procedures of these notes were run in a single driver: 32 `malloc` and 32 `free` calls were counted; in the detailed test both numbers were 2116. There is no leak.

12.4.7 Code quality

The procedures of a threaded tree are short but error-prone. They illustrate a few principles of good code concretely.

Where to put the NULL check. `insert_right` does not check whether `parent` or `child` is `NULL`. This is not an oversight but a design decision. The reasoning given in the lecture is this: `insert_right` is written not to be used on its own but inside another procedure; the `NULL` check should be made in the procedure that uses it. For example, a procedure that inserts into a threaded binary search tree first finds whether the new value is to be inserted to the right or to the left. In

place of the single assignment of an ordinary tree, it calls `insert_right` or `insert_left`. At that point `parent` is already a real node. But if the procedure is to be part of a library and anyone can call it directly, the check should go into the procedure. If no check is made, this should be written in the comment: "`parent` and `child` must not be `NULL`". A defensive wrapper for a general-purpose procedure can be written like this:

```
#include <assert.h>
void insert_right_checked(thread_ptr parent, thread_ptr child) {
  // parent and child must not be NULL; child must not be in a tree yet
  assert(parent != NULL && child != NULL);
  insert_right(parent, child);
}
```

`assert` stops the program, with the file name and line number, if its condition is false. It catches programmer errors early and is very valuable during testing. It is removed in a build compiled with `NDEBUG` defined; so it is used not for errors that user input could cause but for situations of the "this must never happen" kind. For user errors, returning an error code is more appropriate.

Comments. A comment should say what the code intends to do and should not repeat the code itself.

```
/* BAD: repeats the code */
if (!child->right_thread)  // right thread FALSE?
/* GOOD: states the intent */
if (!child->right_thread)  // did parent have a subtree?
```

A comment like "is the right thread `FALSE`" tells the reader nothing new; the same information is written in the code. "Does the parent have a subtree" states the intent. With intent and code side by side, the reader can compare the two and say: "what it wants to do is right, but the code is wrong." Debugging becomes easier this way. The lecture stresses this point: a later reader should not say "I already know that"; comments should say something different from what the code says.

One procedure. It was possible to write a separate procedure for each case: `insert_right_no_subtree`, `insert_right_with_subtree`, `insert_left_no_subtree`, `insert_left_with_subtree`. That would fragment the code; the user would have to decide each time which procedure to call. Reducing four procedures to two without adding complexity is good. But the opposite extreme must be avoided as well. Squeezing all the cases into one procedure with nested `if`s produces unreadable code. In that situation separate procedures are better. The success of `insert_right` comes from defining the common work of the two cases in a common form: copying the thread flag from the parent covers both cases in a single assignment. The method can be summarized like this: first analyze the three or five different cases; then determine the name, return value, and parameters of the procedure; write the common work in a common form. The result is simple, readable, and easily maintained code. The lecture regards these principles as the foundation of software engineering and reminds us that learning them takes years.

Vigilance. When writing threaded tree code one must ask on every line: "I am doing this, but am I doing it correctly?" The code for Case 1 is flawless for Case 1 and silently corrupts the tree in Case 2. After the first six assignments of `insert_right` the tree looks correct, but one thread is still wrong. Such errors are caught only by trying every case separately and by traversing both forward and backward.

12.5 Gains, costs, and a complexity summary

What a threaded tree gives and takes away can be collected in two lists. Gains: the traversal is done without a stack and with *O(1)* extra memory; the move from any node to its successor or predecessor takes at most *O(h)* and amortized *O(1)* within a traversal; the empty fields are not wasted. Costs: two flags are kept in every node; insertion and deletion become more intricate; every procedure that touches the tree has to respect the threads. Table 12.13 gives the costs of the procedures of this chapter together.

Operation	Time	Extra memory	Description
`insucc`	O(h) worst case	O(1)	O(1) if there is a right thread
`inpred`	O(h) worst case	O(1)	O(1) if there is a left thread
`tinorder`	Θ(n)	O(1)	at most 2n + 1 pointers; no stack or recursion
threaded preorder traversal	Θ(n)	O(1)	with the same threads
`insert_right` / `insert_left`	O(h)	O(1)	successor or predecessor search; O(1) in Case 1
`threaded_tree_free`	Θ(n)	O(1)	freeing by traversal
Morris traversal (flagless)	Θ(n)	O(1)	temporary threads; the tree changes during the traversal
recursive inorder (for comparison)	Θ(n)	O(h)	call stack

Table 12.13: Complexity summary. *n* is the number of nodes and *h* is the height of the tree.

12.6 Common mistakes

1. **Treating a pointer as a real child without looking at the flag.** In a threaded tree no pointer is `NULL`. A loop that descends until `NULL` never stops; it goes around the ring through the threads and the head. Every descent is checked with `left_thread` or `right_thread`.
2. **Forgetting the header node.** In a tree without a header, the left thread of the first node and the right thread of the last node are left dangling. If `NULL` is written into them, the traversal procedures need special cases; if nothing is written, garbage values are followed.

3. **Not updating the left thread of the successor in the second case of right insertion.** Forward traversal still gives the right result and the bug stays hidden; a procedure that looks for a predecessor or traverses backwards goes to the wrong node.
4. **Writing `child->right_thread` as a constant `TRUE`.** In the second case the right subtree of the parent is treated as a thread, and the left part of that subtree drops out of the traversal. In the example, E is lost.
5. **Forgetting the `left_thread` of the header in an empty tree.** In an empty tree the left of the header is itself and its flag is `TRUE`. A search that takes `L->left_child` for the root without looking at this flag reads the data of the header as if it were a real value.
6. **Mixing up upper and lower case.** C is case-sensitive: `parent->Right_child` or `parent->Right_thread` does not compile, because the field of the structure is `right_child`. Note also that the threaded structure uses `left_child` while the ordinary structure uses `left`.
7. **Breaking the order of the assignments.** If the line `parent->right_child = child` is written before `child->right_child = parent->right_child`, the old successor of the parent is lost and the child points to itself.
8. **Calling `free` first when freeing the tree.** If the successor is looked up after the node has been freed, freed memory is read. First the successor is found, then the node is freed.

12.7 Exercises and solutions

12.7.1 Drawing the threads of a tree

Question. Draw the threads of the tree A(B(D, E), C(–, F)); how many threads are there?

Solution. The inorder sequence is `D B E A C F`. The tree has 6 nodes; by Theorem 12.1 there are $6 + 1 = 7$ empty fields, that is, 7 threads. The threads were given one by one in Example 12.2: D → head, D → B, E → B, E → A, C → A, F → C, F → head. Two of them go to the head. The solution was also checked with a program that builds the tree and counts the flags.

12.7.2 Finding the predecessor

Question. Write the procedure `inpred` that finds the inorder predecessor.

Solution. The procedure is the mirror image of finding the successor and was given in the section on inserting on the left. If the left flag is `TRUE`, the left pointer points directly to the predecessor. Otherwise we start from the left child and go right until the right flag is `TRUE`. In the running example the predecessor of A is found like this: `p` = B; the right flag of B is `FALSE`, `p` = E; the right flag of E is `TRUE`; the result is E.

12.7.3 Traversing backwards

Question. Write the procedure that traverses a threaded tree backwards in inorder (`G C F A …`).

Solution. Start from the header and move from predecessor to predecessor until we return to the header:

```
void reverse_tinorder(thread_ptr L) {
  thread_ptr p = L;
  for (;;) {
    p = inpred(p);
    if (p == L) break;
    printf("%c ", p->data);
  }
}
```

The predecessor of the header is the last node (G); the predecessor of the first node (H) is the header. In the running example the output is `G C F A E B I D H`. The time is $\Theta(n)$ and the extra memory is $O(1)$; the justification is the mirror image of Theorem 12.3.

12.7.4 The first node of an empty tree

Question. Is it correct to insert the first node into an empty tree (only the header) with `insert_left(head, x)`?

Solution. It is correct. The left pointer of the header points to itself and its left flag is `TRUE`. `insert_left` carries these two over to the child: the left of the child points to the head and is a thread. The right of the child also points to the parent, that is, the head, and is `TRUE`. The left of the header is redirected to the child and its left flag becomes `FALSE`. Because the left flag of the child is `TRUE`, the `if` block does not run. In the end both threads of the single node point to the head; this is exactly the arrangement in Definition 12.3.

12.7.5 Why postorder is hard

Question. Why is postorder traversal hard with threads? What extra information makes the job easier?

Solution. The postorder successor of a node depends on its parent. If the node is the right child of its parent, or the parent has no right subtree, the successor is the parent; otherwise the successor is the first node in postorder in the right subtree of the parent. In both cases we must know who the parent is and which of its children the node is. Threads point to the inorder predecessor and successor; these do not always coincide with the parent. Adding a parent pointer to each node supplies this information directly, and the postorder successor is found in a constant number of steps and one descent.

12.7.6 Inserting into a threaded binary search tree

Question. Write the insertion procedure for a threaded binary search tree using `insert_right` and `insert_left`.

Solution. The search is as in an ordinary binary search tree; the only difference is that we stop when the flag of the direction to descend is `TRUE`. At that point `insert_left` is called if the sought value is smaller than the node, and `insert_right` if it is larger. The new node is always a leaf; so both procedures run in Case 1 and finish in constant time.

```
int threaded_bst_insert(thread_ptr L, char x) {
  // insert x and return 1; return 0 if present
  thread_ptr p;
  if (L->left_thread) {                  // tree is empty
    insert_left(L, new_node(x));
    return 1;
  }
  p = L->left_child;                      // start at the root
  for (;;) {
    if (x == p->data) return 0;
    if (x < p->data) {
      if (p->left_thread) break;         // no left child
      p = p->left_child;
    } else {
      if (p->right_thread) break;         // no right child
      p = p->right_child;
    }
  }
  if (x < p->data) insert_left(p, new_node(x));
  else             insert_right(p, new_node(x));
  return 1;
}
```

If the tree is empty (the left flag of the header is `TRUE`), the first node is inserted on the left of the header; the previous exercise showed that this is correct. The procedure was called in the order `M F T B H P X K`; then `H` was to be inserted once more and 0 was returned. The inorder traversal wrote `B F H K M P T X` and the reverse traversal wrote `X T P M K H F B`. The search takes *O(h)* and the insertion *O(1)*; the total is *O(h)*.

12.8 Summary

- In a binary tree with n nodes, $n - 1$ of the $2n$ pointers are filled; the remaining $n + 1$ fields are empty.
- In a threaded binary tree an empty left pointer is tied to the inorder predecessor and an empty right pointer to the inorder successor. These pointers are called threads.

- The flags `left_thread` and `right_thread` distinguish real links from threads. If the field order is chosen well, the flags fit into padding bytes on a 64-bit machine.
- The header node collects the two dangling threads; the root is at `L->left_child`, and the right of the header is itself. In an empty tree the left flag of the header is `TRUE`.
- Successor: if the right flag is `TRUE`, the right pointer directly; otherwise the leftmost node of the right subtree. The predecessor is the mirror image.
- Threaded inorder traversal follows at most $2n + 1$ pointers: $\Theta(n)$ time and $O(1)$ extra memory. Preorder traversal can be done with the same threads; postorder needs parent information.
- The Morris traversal traverses without flags, by building and removing threads temporarily, in $O(n)$ time and $O(1)$ memory; because the tree is modified during the traversal, it is not suitable for concurrent reads.
- `insert_right` merges the two cases into one procedure by copying the thread flag from the parent; in the second case the child is made the predecessor of the old successor of the parent. `insert_left` is its mirror image.
- Good code: a clearly defined procedure, comments that state the intent, the NULL check in the right place, and a single procedure in which common work is written in a common form.

12.9 Key terms

null pointer thread threaded binary tree predecessor successor header node inorder preorder postorder traversal Morris traversal call stack skewed tree height amortized analysis self-referential struct binary search tree

---

12.10 Test Yourself

1. A binary tree with 15 nodes is being turned into a threaded tree. How many threads are built, and how many go to the header?

   Answer

   The number of empty fields is $15 + 1 = 16$, so 16 threads are built. Whatever the shape of the tree, only two go to the header: the left thread of the first node and the right thread of the last node.

2. In the running example, which node does `insucc(B)` return? How many times does the loop run?

   Answer

   It returns E. The right flag of B is `FALSE` and `p` becomes E. Because the left flag of E is `TRUE`, the loop is never entered: 0 iterations.

3. In the running example, which node does `presucc(I)` return?

   Answer

   E. Because the left flag of I is `TRUE`, the right threads are followed: I → B. The right flag of B is `FALSE`; the loop ends and the right child of B, E, is returned.

4. `insert_right(E, Y)` is applied to the running example. Which case is it? Where do the threads of Y point, and what is the new inorder sequence?

   Answer

   E is a leaf; it has no right subtree, so this is the first case. The left thread of Y points to E and its right thread to A, the old successor of E. The sequence becomes `H D I B E Y A F C G`.

5. `insert_right(A, Y)` is applied to the running example. Whose thread changes?

   Answer

   This is the second case; the right subtree of A (C) becomes the right subtree of Y. The successor of Y is F, the leftmost node of C. The left thread of F points to Y instead of A. The sequence becomes `H D I B E A Y F C G`.

6. In a left-skewed threaded tree of 1000 nodes, how many pointers does `insucc(head)` follow? How many does the whole traversal follow?

   Answer

   The first call descends from the header to the root and then, through 999 left links, to the bottom node: together with the read in line 3, 1001 pointers. The whole traversal follows $2n + 1 = 2001$ pointers; this is the case that reaches the bound of Theorem 12.3 exactly.

7. How many temporary threads does the Morris traversal build on the tree A(B(D, E), C)? What does this number equal in general?

   Answer

   Two: E → A and D → B. One thread is built for each node with a left child, and the same thread is removed later.

8. In `insert_right`, if line 10 (`parent->right_thread = FALSE`) is deleted, in which case does the

tree break?

Answer

In the first case, but insidiously. The flag of the parent stays `TRUE`. Inorder traversal still gives the right result, because the right pointer taken for a "thread" already points to the child, that is, to the successor of the parent. The other procedures skip the child. When C is inserted on the right of the tree A(B, –), the inorder traversal writes `B A C`; the preorder traversal writes `A B` and the backward traversal writes `A B`. C is lost. In the second case no harm is done, because the flag is already `FALSE`.

12.11 Further reading

- E. Horowitz, S. Sahni, and S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2nd ed., Section 5.5. The structure of these notes, the `insert_right` procedure, and the layout of the header node follow the treatment there.
- D. E. Knuth, *The Art of Computer Programming*, Volume 1, Section 2.3.1. Threaded trees, finding the successor, and the algorithms for insertion into a threaded tree are presented with proofs and exercises.
- J. M. Morris, "Traversing binary trees simply and cheaply", *Information Processing Letters* 9(5), 1979. This is the short original paper on flagless traversal with temporary threads.

12.12 References

1. E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2nd ed., Silicon Press, 2008. Section 5.5, Threaded Binary Trees.
2. A. J. Perlis, C. Thornton, "Symbol manipulation by threaded lists", *Communications of the ACM* 3(4), 1960.
3. J. M. Morris, "Traversing binary trees simply and cheaply", *Information Processing Letters* 9(5), 1979.
4. D. E. Knuth, *The Art of Computer Programming*, Volume 1: Fundamental Algorithms, 3rd ed., Addison-Wesley, 1997. Section 2.3.1.

Priority Queues and Heaps

In an ordinary queue, items leave in the order in which they arrived. In many real systems, however, the item to handle first is not the one that arrived first but the one that is most urgent: the aircraft with the least fuel left, the shortest job, the nearest vertex. These notes define the operations of such a queue and implement it with a structure called a heap. A heap is a complete binary tree stored in an array. It uses no pointers, yet it behaves like a tree. The notes first set up the index rules of this array. They then write the insertion and deletion routines in C, trace them line by line, and justify their costs. A cost table and real measurements show why a heap beats simple structures such as arrays and linked lists. The last sections show how to turn an array into a heap in linear time, describe the d-ary heap and heap sort, and survey where priority queues are used.

LEARNING OUTCOMES

Defining
Define a priority queue and a max/min heap; distinguish a heap from a binary search tree.

Setting up the representation
Store a complete binary tree in a 1-based array: parent `i/2`, children `2i` and `2i+1`.

Insertion and deletion
Trace the routines `insert_max_heap` and `delete_max_heap` step by step.

Analyzing
Show why insertion and deletion take $O(\lg n)$ and why the best case is $\Omega(1)$.

Comparing
Compare a heap with unsorted/sorted arrays and lists using a cost table.

Extending
Explain why bottom-up heap construction is $O(n)$; recognize the d-ary heap.

13.1 Priority queues

A queue is a structure in which items enter at one end and leave at the other. This discipline is fair: first come, first served. But it is not the right discipline for every job. In a hospital emergency room, patients are not examined in order of arrival but according to the severity of their condition. This section describes the data structure that captures such a discipline and explains why we need it.

13.1.1 How it differs from a queue

An ordinary queue has two operations. Insertion puts the new item at the back of the queue; deletion takes the item at the front. This rule is called **first in, first out** (FIFO). In a priority queue, by contrast, every item has a **priority**, and deletion looks at that priority.

Definition 13.1

A **priority queue** is a queue in which every item carries a priority value, and the item to be removed first is determined by its priority value rather than by its arrival order. In a **max priority queue** the item with the highest value is handled first; in a **min priority queue** the item with the lowest value is handled first.

Most sources call the priority value the **key**. These notes use the two words interchangeably. Which value counts as "high priority" depends on the problem: in a race the shortest time is best; in an auction, the highest bid. The difference between the two kinds lies only in the direction of the comparison. For this reason the notes mostly describe the max priority queue; the min version is obtained by reversing the comparison signs.

Figure 13.1 shows the difference on four items. Each item is named by a letter and has a number as its priority: *A:3*, *B:9*, *C:5*, *D:7*. The four items arrived in this order. The ordinary queue yields the first arrival, *A*. The max priority queue yields *B*, which has the highest priority; *B* does not wait its turn even though it came second. On the second deletion the ordinary queue yields *B* and the priority queue yields *D*. The complete output order of the priority queue is *B, D, C, A*: items leave in decreasing order of priority.

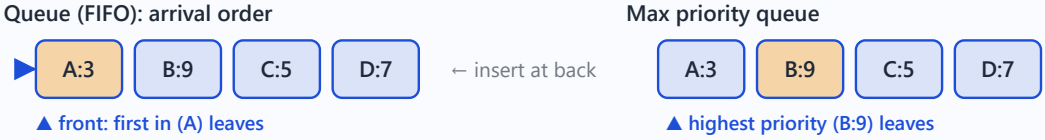


Figure 13.1: The same four items in two structures. On the left, an ordinary queue: insertion is at the back, and the first arrival *A* leaves. On the right, a max priority queue: the item with the highest priority, *B:9*, leaves.

The definition says nothing about how the items are arranged inside the structure. It says only which item must come out first. This is an important freedom. The structure that implements the queue may arrange its items however it likes, as long as it can find and return the highest-priority item on demand. Most of these notes describe the arrangement that makes the best use of this freedom: the heap.

> **Common mistake**
>
> Believing that a priority queue always keeps its items sorted. The definition only determines which item comes out first. Keeping the items sorted is one way to satisfy that condition, but not the cheapest. A heap keeps only the highest-priority item ready, without sorting the rest completely; that is exactly why it is cheap.

13.1.2 Why we need it

The slides motivate the priority queue with two examples. In both, arrival order does not lead to the right decision.

The photocopier. Everyone who uses a photocopier pays a fixed fee. A user who copies three pages pays the same as one who copies thirty. The owner wants to maximize earnings, that is, to serve as many users as possible in a given time. The way to do that is to give priority to the user with the shortest usage time. If the three-page job is done first, that user leaves at once and pays; even if the thirty-page job waits, total earnings rise. Here the item to handle first is the one with the smallest value: this is a **min** priority queue.

The airport. Consider aircraft waiting for a slot to land. Landing the aircraft that arrived first at the tower looks fair at first sight. But one cannot tell an aircraft that is almost out of fuel, “you are at the back of the queue, wait your turn.” Landing clearance goes to the aircraft with the least fuel left. This too is a min priority queue: the key is the amount of remaining fuel.

Real systems work this way too. An operating system keeps the processes that are ready to run in a priority queue. When an interrupt arrives or a process's time slice expires, the highest-priority process in the queue is given the processor. In such systems the queue changes constantly: new jobs arrive and finished jobs leave. The structure therefore has to make both insertion and removal of the highest-priority item fast. The real difficulty is keeping these two operations cheap together.

13.1.3 The abstract data type

A type whose operations and behavior are specified but whose implementation is hidden is called an **abstract data type** (ADT). The priority queue is an ADT too. The slides give the interface of the max priority queue as follows:

```c
/* max priority queue interface */
bool is_empty(void);
bool is_full(void);
void insert(element item);     /* O(lg n) */
bool pop_max(element *out); /* remove + return */
bool peek_max(element *top);  /* look, no removal */
```

The interface has five operations. `is_empty` and `is_full` query the state of the queue. In an implementation with fixed capacity, nothing can be inserted into a full queue, which is why `is_full` is part of the interface. `insert` puts an item, together with its priority, into the structure. `pop_max` both returns the highest-priority item and removes it from the structure; the slides call this operation *delete*. `peek_max` reads the highest-priority item without removing it.

The pop and peek operations hand back the item through a pointer parameter, not as the return value. The return value is a `bool` that reports whether the operation succeeded. An empty queue has nothing to remove. A function that returned the item would have to invent a "meaningless" item in that case. Returning `false`, in contrast, tells the caller plainly that there is no item. For a function that returns the item as a structure, there is no suitable value to signal the "empty" state either.

The interface does not say how the queue is to be implemented. A priority queue can be implemented with an unsorted array, a sorted array, a linked list, or a heap. All of them offer the same interface, and the program that uses them cannot tell the difference. The difference lies only in cost. Simple structures make one operation very cheap and the other very expensive. A heap does both in *O(lg n)* time. This balance makes the heap the most common implementation of the priority queue.

13.2 Heap structure

A heap combines two rules. The first rule fixes the *shape* of the structure: a heap is a complete binary tree. The second fixes the *order* of the values: every node is greater than or equal to its children. This section first describes the shape rule and how it is stored in an array, then the order rule.

A note on terminology: do not confuse the data structure called a heap with the *heap memory* from which `malloc` allocates. The two have nothing in common except the name. In the C code of the slides, the array that holds the heap is called `heap`, and the routines have names such as `insert_max_heap`; the notes use the same names.

13.2.1 A complete binary tree in an array

In a binary tree every node has at most two children. The node directly above a node is its **parent**. Nodes at the same distance from the root are on the same **level**. In these notes the root is on level 0.

Definition 13.2

A **complete binary tree** is a binary tree in which every level except possibly the last is full, and the nodes of the last level are packed from left to right without gaps.

The shape of a complete binary tree depends only on the number of nodes. A complete binary tree with 6 nodes can be drawn in only one way: the root, its two children, and three nodes at the last level, filled from the left. This rigidity pays off. If the nodes are numbered level by level, and left to right within each level, the number of every node is completely determined by the shape of the tree. The root gets number 1, its left child 2, its right child 3; the nodes of the next level continue as 4, 5, 6, 7. These numbers can be used directly as array indices.

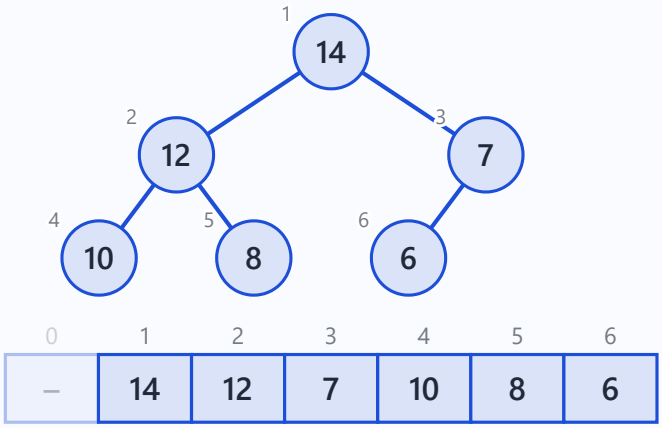


Figure 13.2: The max heap *14, 12, 7, 10, 8, 6*. Above, the tree view; the small numbers next to the nodes are indices. Below, the layout of the same heap in the array; cell 0 is unused.

Figure 13.2 shows a complete binary tree with six nodes and its array counterpart. Each node of the tree sits in the array cell whose index equals its number. The 14 at the root is in `heap[1]`, the root's children 12 and 7 are in `heap[2]` and `heap[3]`, and the 10, 8 and 6 on the last level are in `heap[4]`, `heap[5]` and `heap[6]`. Cell 0 of the array is not used. This small waste simplifies the index formulas.

The most important consequence of the numbering is that the relationship between a parent and its children is found by simple arithmetic. The left child of node i is at index $2i$, the right child at $2i + 1$, and the parent at $\lfloor i/2 \rfloor$. Because integer division in C rounds down, the expression `i / 2` gives this floor directly. Table 13.1 collects these rules.

Node	Index (1-based)	Index (0-based)
root	`1`	`0`
parent of *i*	`i / 2`	`(i - 1) / 2`
left child	`2i`	`2i + 1`
right child	`2i + 1`	`2i + 2`

Node	Index (1-based)	Index (0-based)
last item	`n`	`n - 1`

Table 13.1: Index rules of a complete binary tree in a 1-based array (*n* is the number of items).

A small count shows why the rules are correct. Level *d* holds 2^d nodes, and the number of the first node of that level is 2^d: 1, 2, 4, 8, ... Let node number *i* be *k*-th from the left on its level (counting from 0); that is, $i = 2^d + k$. Each of the *k* nodes to its left has two children. So its left child is at position $2k$ from the left one level down. Since the first number on the next level is 2^{d+1}, the number of the left child is $2^{d+1} + 2k = 2(2^d + k) = 2i$. The right child is immediately to its right: $2i + 1$. The formula for the parent is the inverse of these. Both $2i$ and $2i + 1$ give *i* when divided by 2 and rounded down.

Some sources start the array at cell 0. Then every number drops by one and the formulas change: the parent is `(i - 1) / 2` and the children are `2i + 1` and `2i + 2`. Both layouts are correct, but a program must not mix them. The slides and these notes use the 1-based layout, which keeps the formulas simple.

Example 13.1

In a 1-based heap, the parent of node 13 is node $\lfloor 13/2 \rfloor = 6$. Its left child is node $2 \cdot 13 = 26$ and its right child is node 27. These children exist only if the heap has enough items: the left child exists if $n \geq 26$, the right child if $n \geq 27$. Node 13 is on level 3, because $8 \leq 13 < 16$; it is the 6th node from the left on its level ($13 = 8 + 5$, which is 5 counting from 0). If the same tree were stored in a 0-based array, this node would be at `a[12]`; its parent would be $(12 - 1)/2 = 5$ and its children 25 and 26. The same nodes are found in both layouts; only their numbers shift by one.

This representation costs nothing. In a linked tree every node carries two child pointers; for an `int` key that means 8 or 16 extra bytes per node. In the array these pointers are unnecessary, because a complete binary tree has no gaps between nodes and the positions of the children are computed. Since the array cells sit side by side in memory, the processor cache also works well with this structure. The gain is specific to complete binary trees. A sparse, unbalanced tree stored this way would leave most of the array empty. That is exactly why the shape rule exists in a heap.

Common mistake

Mixing up the 1-based and 0-based formulas. Suppose the array starts at cell 0 and the parent is computed as `i / 2`. Then the parent of cell 1, which is thought of as a child of the root, is still cell 0, but the parent of cell 2 comes out as cell 1. In the 0-based layout, however, cell 2 is the right child of the root. As a result the wrong nodes are compared and the heap silently breaks. The program does not crash; it just returns the wrong item. Such a bug is hard to find.

13.2.2 A heap is not a binary search tree

The word heap should bring to mind a complete binary tree, not a binary search tree. Both structures are binary trees and in both the values obey some order, but the orders are very different. In a **binary search tree** the values in the left subtree of every node are smaller than the node, and those in the right subtree are larger: *left < root < right*. In a heap there is only the rule *parent ≥ child*. No order is required between the left child and the right child.

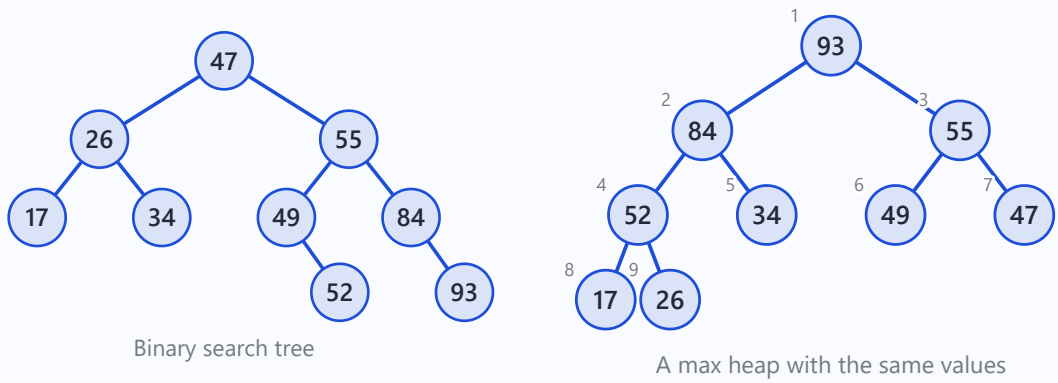


Figure 13.3: The same nine values in two structures. On the left, a binary search tree: each node has smaller values on its left and larger on its right; the tree is not complete. On the right, a max heap: a complete binary tree in which every parent is larger than its children.

Figure 13.3 shows how the same nine values, *47, 26, 55, 17, 34, 49, 84, 52, 93*, are placed in the two structures. In the binary search tree the root is 47; 26 is on the left and 55 on the right. The shape of the tree depends on the order in which the values arrived. In the example, 49 and 84 each have a node below them while 17 has no children at all; the tree is not complete. In the heap the root is 93, the largest value. The shape is complete: the first three levels are full and the last level has two nodes from the left. The left child of the root is 84 and the right child is 55; there is no ordering rule between these two values.

The two structures answer different questions. In a binary search tree, searching for a value is easy: at each node one goes left or right. In a heap there is no such direction; to find 34 one may have to look in both subtrees of the root. In return, the one question a heap answers is answered very fast: the largest value is always at the root. A heap is not a search structure but a priority queue structure.

13.2.3 Max heaps and min heaps

Definition 13.3

A **max heap** is a complete binary tree in which the value of every node is not smaller than the values of its children. A **min heap** is a complete binary tree in which the value of every node is not larger than the values of its children. This condition on the values is called the **heap property**.

The phrase "not smaller than" in the definition is chosen deliberately. A parent may be equal to its child; two items with the same priority may stand in a heap as parent and child. The two parts of the definition are not sufficient separately. A tree that satisfies the heap property but is not complete is not a heap; nor is a complete tree that violates the heap property. These definitions are given to implement the max and min priority queues: a max heap implements a max priority queue, and a min heap implements a min priority queue.

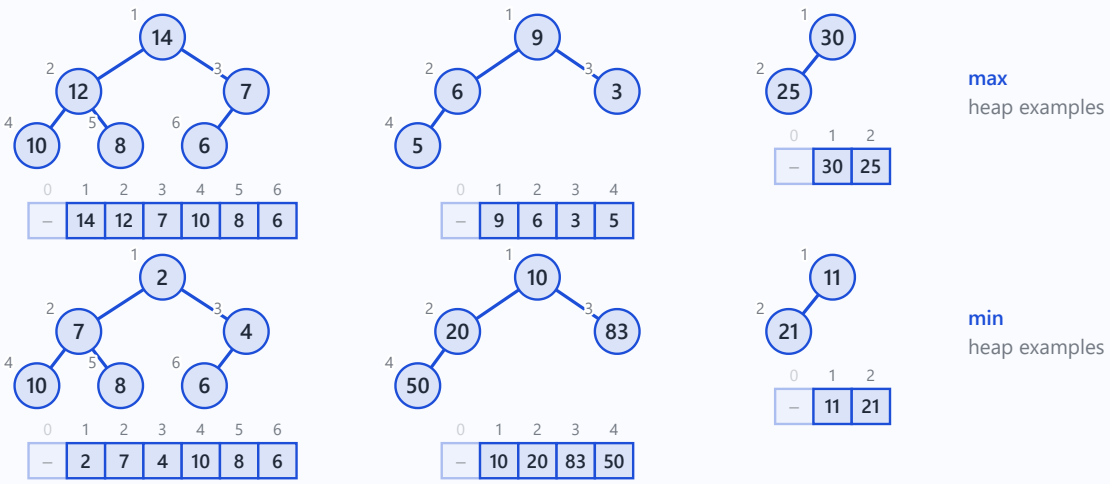


Figure 13.4: Three max heaps above and three min heaps below, each with its array representation underneath. Heaps with two items are valid too.

Figure 13.4 gives three examples of each kind. The first max heap above is the main example of the notes: *14, 12, 7, 10, 8, 6*. Every parent is larger than its children: $14 \geq 12$, $14 \geq 7$, $12 \geq 10$, $12 \geq 8$, $7 \geq 6$. The right child of the root, 7, is smaller than 10 and 8 in the left subtree. This is not a problem; 7 and 10 have no parent–child relationship. In the min heaps below, the rule is reversed. In the first example, *2, 7, 4, 10, 8, 6*, every parent is smaller than its children and the smallest value, 2, sits at the root.

The heap property links only adjacent levels. But chained together it gives a much stronger result: on any path from the root to a leaf, the values form a non-increasing sequence. From this it follows that the root holds the largest value of the whole tree.

Theorem 13.1 (The item at the root)

In a non-empty max heap, the value of every node is not larger than the value of any of its ancestors. In particular, the value at the root is the largest value in the heap.

Proof. Take a node numbered i. Going to its parent $\lfloor i/2 \rfloor$, then that node's parent $\lfloor i/4 \rfloor$, and so on, we reach the root, number 1, in finitely many steps, because each step at least halves the number. Every consecutive pair in this chain is a parent–child pair, and by the heap property the parent's value is not smaller than the child's. Chaining the inequalities gives $heap[1] \geq \ldots \geq heap[\lfloor i/2 \rfloor] \geq heap[i]$. Since i was an arbitrary node, the root is not smaller than any value. ■

The theorem answers the most frequent question about a priority queue, "which item is the largest?", in $O(1)$: it is `heap[1]`. The rest of the array is not sorted. In the example heap, the second largest value, 12, is at index 2, but the third largest, 10, is at index 4, not index 3. The guarantee holds only for index 1. The second largest value is always one of the root's children. The third largest may be the root's other child, or a child of the larger child.

13.2.4 Heap or not?

To tell whether an array is a max heap, it is enough to look at all parent–child pairs. In the array representation the shape rule holds automatically, because indices $1..n$ are filled without gaps. Only the order rule remains: for every $k \geq 2$ we need `heap[k/2] ≥ heap[k]`. No order is required between siblings or cousins. The following helper function performs this check:

```
bool is_heap(element a[], int n) {
  for (int k = 2; k <= n; k++)
    if (a[k / 2].x < a[k].x) return false;  /* parent < child */
  return true;
}
```

The loop compares each node from 2 to n once with its parent. Since every node other than the root has exactly one parent, all parent–child pairs are checked exactly once. It returns `false` at the first violation. The cost is at most $n - 1$ comparisons, that is, $O(n)$. This function is not part of the heap, but it is very useful for testing the routines we write. All the numbers in these notes were verified with test programs that run this check after every operation.

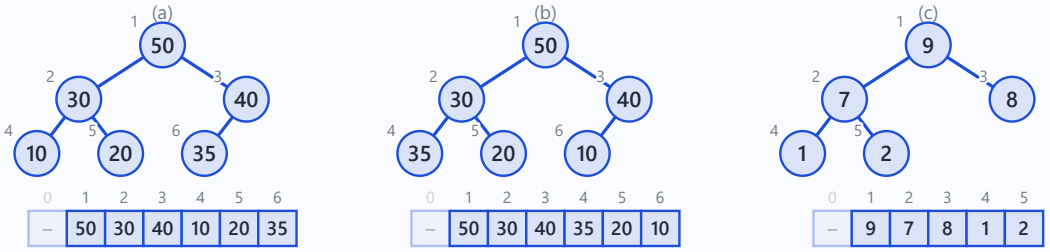


Figure 13.5: Three candidate arrays. Which one is a max heap?

Example 13.2

Figure 13.5 shows three arrays. (a) *50, 30, 40, 10, 20, 35* is a max heap. The parent–child pairs are *50 ≥ 30*, *50 ≥ 40*, *30 ≥ 10*, *30 ≥ 20* and *40 ≥ 35*; all hold. It does not matter that 35 is larger than 30; the parent of 35 is 40.

(b) *50, 30, 40, 35, 20, 10* is not a max heap. The parent of 35 at index 4 is 30 at index *⌊4/2⌋ = 2*, and *30 < 35*. `is_heap` stops at $k = 4$ on this array and returns `false`.

(c) *9, 7, 8, 1, 2* is a max heap. The tree is complete and every parent is larger than its children: *9 ≥ 7*, *9 ≥ 8*, *7 ≥ 1*, *7 ≥ 2*. It does not matter that 1 and 2 are smaller than 8, that is, that a value in the right subtree is larger than values in the left one. `is_heap` returned `true`, `false`, `true` for the three arrays, respectively.

13.2.5 C definitions

The first implementation in the slides keeps the heap in a global array with an item counter. The definitions are as follows:

```
#define MAX_ELEMENTS 200   /* max items + 1 */
#define HEAP_FULL(n) ((n) == MAX_ELEMENTS - 1)
#define HEAP_EMPTY(n)  (!(n))

typedef struct {
  int x;          /* priority (key) */
  /* other fields */
} element;

element heap[MAX_ELEMENTS];  /* [0] unused */
int n = 0;                   /* item count */
```

`MAX_ELEMENTS` is the size of the array. Because cell 0 is unused, the heap holds at most *MAX_ELEMENTS − 1 = 199* items; hence the “max items + 1” in the comment. The indices in use are 1..199. The macro `HEAP_FULL(n)` tests whether the item count has reached this limit, and

`HEAP_EMPTY(n)` tests whether the item count is zero. The expression `!(n)` yields 1 if *n* is zero and 0 otherwise.

The item type `element` is a structure. Its only field, `x`, holds the priority. In a real application the structure would hold other fields too: a process number, an aircraft's flight code. The heap routines look only at `x`; the other fields travel with the item. For this reason the routines copy items as whole structures, not as integers. The assignment `heap[i] = heap[j]` copies all fields of the structure at once. In the examples of these notes the structure holds a single `int`, so it is 4 bytes.

Parenthesizing the parameter in a macro is not a habit but a necessity. A macro is not a function; the preprocessor substitutes the text as it is. Suppose `#define HEAP_EMPTY(n) (!n)` had been written. The call `HEAP_EMPTY(a + b)` expands to `(!a + b)`. Since the `!` operator is applied before `+`, this expression is evaluated as *(!a) + b*; for `a = 0`, `b = 5` the result is 6, which means “empty”. With the correct definition the expression becomes `(!(a + b))` and yields 0. The outer parentheses serve the same purpose: when the result of the macro is used inside a larger expression, it must stay together as a whole.

Common mistake

Defining the array as `element heap[MAX_ELEMENTS - 1]`, that is, with 199 items (indices 0..198). `HEAP_FULL` considers the heap full only when *n = 199*. At the 199th insertion, *n* rises to 199 and the item is written to `heap[199]`, a cell outside the array. This bug was tried with a test program: the 199th insertion wrote to the memory just after the array and the program went on without any warning. The undefined-behavior sanitizer (`-fsanitize=undefined`) stopped only when the same cell was read, with the message “index 199 out of bounds for type 'element[199]'”. The size of the array must be `MAX_ELEMENTS`.

A global array and a global counter are a simple start, but they have limits. The program can have only one heap. The capacity is fixed at compile time. The routines do not take the array they work on as a parameter. These drawbacks can be removed by using a structure. But the logic of the routines is easiest to see on the global array; insertion and deletion are therefore written in that form first.

13.3 Insertion

When a new item is inserted into a heap, two rules must be preserved at once: the tree must stay complete, and no parent may be smaller than its children. Insertion handles the two rules in turn: it fixes the shape first, then the order.

13.3.1 Make room first, then sift

When a node is added to a complete binary tree, where the new node goes is already determined: the first empty position on the last level. If the last level is full, a new level is opened and its leftmost position is used. In the array representation the index of this position is *n + 1*. So the position to open for the new item is known exactly. This position is opened first; the tree is complete again.

But the new item may be larger than its parent. Then the heap property is broken at this parent–child pair. To repair it, the new item is moved upward to its proper position. If the parent is smaller than the new item, the parent moves down one level and the candidate position for the new item moves up one level. This continues until a parent not smaller than the new item, or the root, is reached. This upward movement is called **sifting up** (up-heap).

Figure 13.6 shows the insertion of 15 into the heap *14, 12, 7, 10, 8, 6* in five moments. Initially the heap has six items. The first empty position is 7; it is the right child of node 3, the node holding 7. The new item, 15, is larger than 7, so 7 is shifted down one level to position 7 and the hole moves up to position 3. The parent of position 3 is the 14 at the root. 15 is larger than that too; 14 goes down to position 3 and the hole moves up to position 1, the root. The root has no parent. 15 is written at the root and the heap property is restored. The final state is *15, 12, 14, 10, 8, 6, 7*.

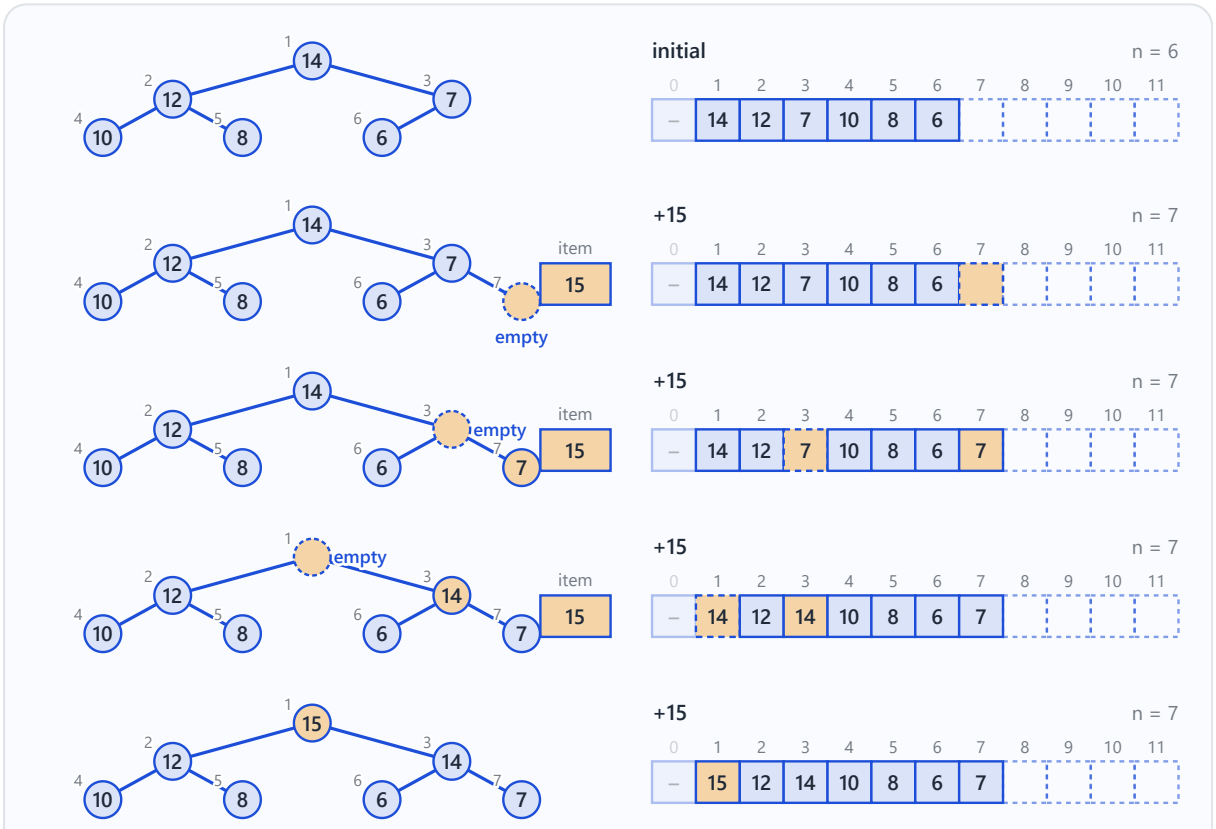


Figure 13.6: Inserting 15. From top to bottom: the initial state; opening the first empty position (7); shifting 7 down; shifting 14 down; writing 15 at the root. The orange circle shows the hole, and the box on the right shows the `item` waiting to be placed.

Note that the movement takes place along a single path. The path from the new position to the root is 7 → 3 → 1. The rest of the heap, for example the subtree of 12, is not touched at all. This is the source of the cheapness of insertion. The length of the path equals the height of the tree, and the height of a complete binary tree grows with the logarithm of the number of items.

Why does sifting up preserve the heap property? Every parent that moves down along the path is smaller than the new item. The children of the position it moves to were already its grandchildren in its old position; by the heap property they are not larger than it. So the parent that moves down obeys the rule in its new place as well. The new item, where it stops, is not larger than its parent. Nor is it smaller than its children: one of them is the parent that it displaced downward, and the other is the old child of that parent. Both are smaller than the new item. Thus at the end of the insertion all parent–child pairs obey the rule again.

13.3.2 `insert_max_heap`: shifting, not swapping

The insertion routine in the slides is as follows:

```
void insert_max_heap(element item) {
    if (HEAP_FULL(n)) {
    heap_full();             /* error message */
    return;
  }
  int i = ++n;
  while (i != 1) {            /* i == 1: at the root */
    if (item.x <= heap[i / 2].x) break;
    heap[i] = heap[i / 2];  /* move parent down */
    i /= 2;                   /* go up a level */
  }
  heap[i] = item;
}
```

The routine first checks whether the heap is full. If it is, `heap_full()` is called and the routine returns without inserting. `heap_full` is a simple routine that prints an error message; the slides do not show its body. The test programs used a one-line function that writes “heap full” to the standard error stream.

The line `int i = ++n;` does two things. `++n` first increments the item count and then yields the incremented value; `i` therefore receives the index of the first empty position, that is, the new last position. After this line `heap[i]` holds no meaningful value yet. The new item is not written there immediately, because that may not be its place. `i` marks the current position of the hole.

The loop condition is `i != 1`. When `i` is 1, the hole has reached the root, and the root has no parent; we cannot go higher. The first statement of the loop compares the new item with the parent of the hole. If `item.x <= heap[i / 2].x`, the parent is large enough, the hole is the place for the new

item, and the loop ends with `break`. Otherwise the parent is copied into the hole, that is, one level down: `heap[i] = heap[i / 2]`. Then `i /= 2` moves the hole up to the parent's old position. When the loop ends, `i` marks the place of the new item and `heap[i] = item` writes the item there.

The idea behind the routine deserves attention: there is no swap. Many sources describe sifting up by writing the new item at the end and then exchanging it with its parent repeatedly. Each exchange needs a temporary variable and three assignments. The **hole method** used here does not carry the new item along the path. Only the larger ancestors are shifted down one level each, and each shift is a single assignment. The new item is written once, at the very end. For an insertion that climbs k levels, swapping makes $3k$ assignments, whereas this method makes $k + 1$. When the item type is a large structure, the difference matters even more.

Common mistake

Writing the loop condition as `i > 0` or `i >= 1`. If the loop is entered with `i` equal to 1, it reads `heap[i / 2]`, that is, the unused `heap[0]`. Cell 0 holds a meaningless value. If that value is smaller than the new item, the assignment `heap[1] = heap[0]` copies the meaningless value into the root, `i` becomes 0, and the new item is written to `heap[0]`, outside the heap. The root is left with a meaningless value and the new item is lost. To stop at the root, the condition must be `i != 1` (or `i > 1`).

13.3.3 Code trace: `insert(15)`

Table 13.2 traces the insertion of Figure 13.6 through the lines of the code. The line numbers follow the trace slide in the slides: line 2 is the full check, line 6 is `int i = ++n;`, line 7 the loop condition, line 8 the comparison, line 9 the shift, line 10 `i /= 2`, and line 12 the final write. Initially the heap is *14, 12, 7, 10, 8, 6* and $n = 6$.

Line	`n`	`i`	`heap[i/2]`	Condition / action	`heap[1..7]`
2	6	—	—	`HEAP_FULL(6)` is false: there is room	14, 12, 7, 10, 8, 6, —
6	7	7	7	`i = ++n`: first empty position	14, 12, 7, 10, 8, 6, _
7	7	7	7	$7 \neq 1$: enter the loop	14, 12, 7, 10, 8, 6, _
8	7	7	7	$15 \leq 7$? No	14, 12, 7, 10, 8, 6, _
9	7	7	7	`heap[7] = heap[3]`	14, 12, _, 10, 8, 6, 7
10	7	3	14	`i /= 2`	14, 12, _, 10, 8, 6, 7
7, 8	7	3	14	$3 \neq 1$; $15 \leq 14$? No	14, 12, _, 10, 8, 6, 7
9	7	3	14	`heap[3] = heap[1]`	_, 12, 14, 10, 8, 6, 7
10	7	1	—	`i /= 2`	_, 12, 14, 10, 8, 6, 7
7	7	1	—	$i = 1$: the root is reached, the loop ends	_, 12, 14, 10, 8, 6, 7

Line	`n`	`i`	`heap[i/2]`	Condition / action	`heap[1..7]`
12	7	1	—	`heap[1] = item`	15, 12, 14, 10, 8, 6, 7

Table 13.2: Trace of `insert_max_heap(15)`. The array column shows `heap[1..7]`; `_` marks the hole.

The “hole” in the table is a notational convenience. In memory that cell is not empty; it holds a copy of the old value. After the line `heap[7] = heap[3]`, the value 7 is in both cell 3 and cell 7. The copy in cell 3 is now meaningless, because it will be overwritten in the next step. The trace therefore shows that cell as a hole. The same happens with 14: after `heap[3] = heap[1]` the 14 is in two cells, and the copy at the root is overwritten by 15 on the last line.

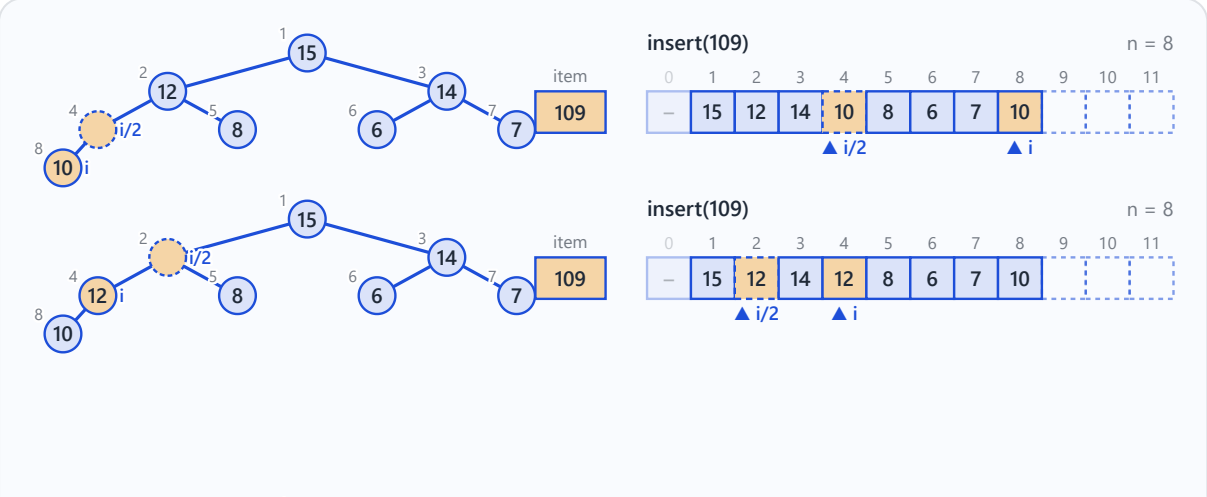
The loop runs fully twice and ends at the third check because `i == 1`. In total there are two comparisons, two shifts and one final write. The new item has climbed two levels along the path. That it stops at the root is no coincidence: 15 is larger than every value in the heap, and by Theorem 13.1 the largest value must end up at the root.

13.3.4 Code trace: `insert(109)`

In the second insertion the heap is *15, 12, 14, 10, 8, 6, 7* with *n = 7*, and 109 is inserted. 109 is larger than all items, so this insertion is a worst case: the item will climb all the way to the root. The last level (4, 5, 6, 7) is full. The new position, 8, is therefore the first position of a new level, and it is the left child of node 4, that is, of 10.

Round	`i`	`i/2`	Comparison	Action	`heap[1..8]` (end of round)
start	8	4	—	`i = ++n = 8`	15, 12, 14, 10, 8, 6, 7, _
1	8	4	*109 ≤ 10*? No	`heap[8] = heap[4]`, `i = 4`	15, 12, 14, _, 8, 6, 7, 10
2	4	2	*109 ≤ 12*? No	`heap[4] = heap[2]`, `i = 2`	15, _, 14, 12, 8, 6, 7, 10
3	2	1	*109 ≤ 15*? No	`heap[2] = heap[1]`, `i = 1`	_, 15, 14, 12, 8, 6, 7, 10
end	1	—	*i = 1*: the loop ends	`heap[1] = item`	109, 15, 14, 12, 8, 6, 7, 10

Table 13.3: Summary of the trace of `insert_max_heap(109)`. Each row is one iteration of the loop.



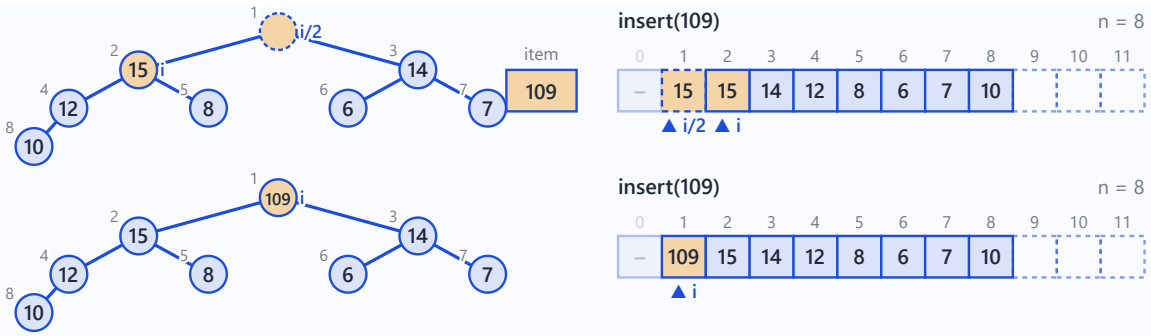


Figure 13.7: Four moments during `insert(109)`. From top to bottom: shifting 10, 12 and 15 down one level each; writing 109 at the root. *i* marks the hole and *i/2* marks the parent being compared.

The loop runs fully three times: the path from the new position to the root is 8 → 4 → 2 → 1, which has three edges. Each round makes one comparison and one shift. At the fourth check `i == 1` and the loop ends without a comparison. The three items on the path, 10, 12 and 15, have each moved down one level. The items off the path, 14, 8, 6 and 7, have not moved. This insertion did the most work possible in a heap of 8 items. A complete binary tree with 8 items has four levels, and the new item climbed from the bottom level to the root.

13.3.5 Repeated insertion

The insertion example in the slides consists of five values: 15, 109, 107, 3 and 15 are inserted in turn into the heap *14, 12, 7, 10, 8, 6*. The first two were traced above. The remaining three insertions behave differently; Table 13.4 gives the path and result of each insertion.

Insertion	New position	Path	Shifts	Heap (after)
+15	7	7 → 3 → 1	2	15, 12, 14, 10, 8, 6, 7
+109	8	8 → 4 → 2 → 1	3	109, 15, 14, 12, 8, 6, 7, 10
+107	9	9 → 4 → 2	2	109, 107, 14, 15, 8, 6, 7, 10, 12
+3	10	10	0	109, 107, 14, 15, 8, 6, 7, 10, 12, 3
+15	11	11 → 5	1	109, 107, 14, 15, 15, 6, 7, 10, 12, 3, 8

Table 13.4: Five consecutive insertions. “Path” lists the indices the hole passes through, from the new position to the position where it stops.

In the insertion of 107, the first empty position is 9; its parent is the 12 at position 4. Since *107 > 12*, 12 shifts down to position 9 and the hole moves up to 4. The parent of 4 is the 15 at position 2; since *107 > 15*, 15 shifts down too and the hole moves up to 2. The parent of 2 is the 109 at the root, and *107 ≤ 109*. The parent is large enough, the loop stops, and 107 is written at position 2. This insertion stopped in the middle of the path without reaching the root.

In the insertion of 3, the new position is 10 and its parent is the 8 at position 5. Since *3 ≤ 8*, the loop ends with `break` at the very first comparison. No shift is made and 3 is written directly at position 10. This is the best case of insertion: one comparison and one write, that is, constant work however

large the heap is.

Finally the second 15 is inserted. The new position is 11; its parent is again the 8 at position 5. Since *15 > 8*, 8 shifts down to position 11 and the hole moves up to 5. The parent of 5 is the 107 at position 2, and *15 ≤ 107*. 15 is written at position 5. The heap now has two 15s: one at position 4 and one at position 5. These two items are siblings and there is no order rule between them. The phrase “not smaller than” in the definition allows items of equal priority.

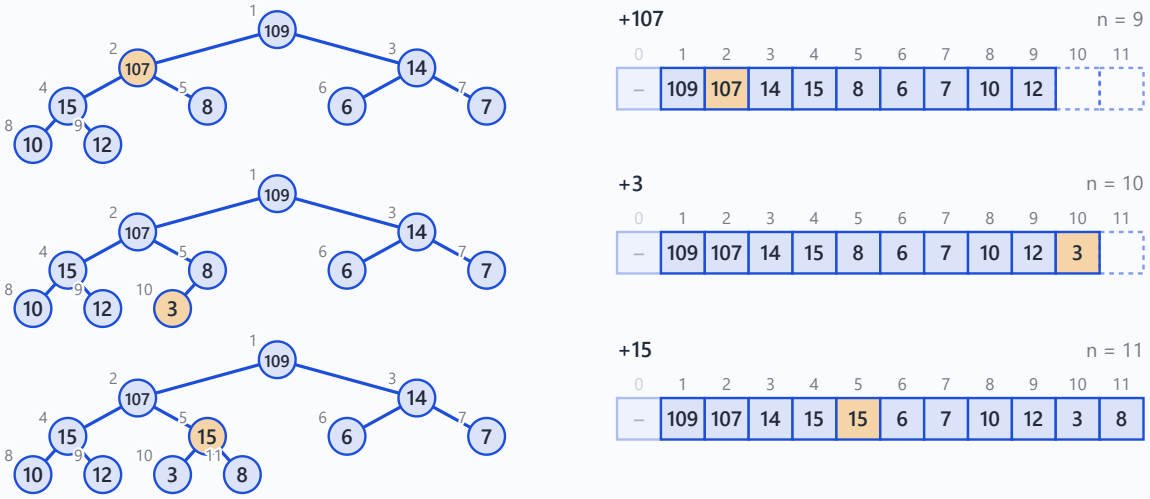


Figure 13.8: Three states after inserting 107, 3 and 15. The orange node is the position where the most recently inserted item came to rest.

Figure 13.8 shows the results of these three insertions. The numbers of shifts in the five insertions are 2, 3, 2, 0 and 1. The insertion of 3 made no shift at all; this is the best case and takes constant time. 109 climbed all the way to the root; this is the worst case and takes time proportional to the height. The cost of the same routine varies between these two extremes depending on the size of the inserted value.

13.3.6 The cost of insertion

The cost of insertion is determined by the number of iterations of the loop. Each iteration does constant work: one comparison, one shift and one division. The number of iterations is the number of levels the hole climbs. The hole can climb from the bottom all the way to the root; so the number of iterations is at most the height of the tree. What remains is to find the height of a complete binary tree.

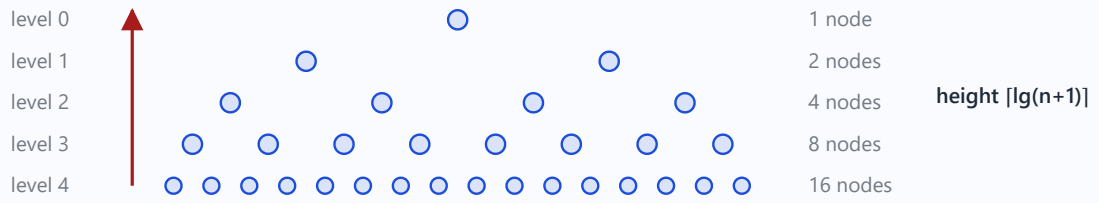


Figure 13.9: The levels of a complete binary tree. Each level has twice as many nodes as the previous one: 1, 2, 4, 8, 16. The arrow shows the longest path along the height.

Theorem 13.2 (Height of a complete binary tree)

A complete binary tree with $n \geq 1$ nodes has $\lfloor \lg n \rfloor + 1 = \lceil \lg(n+1) \rceil$ levels. A path from a leaf to the root has at most $\lfloor \lg n \rfloor$ edges.

Proof. Number the levels of the tree from 0 to h. Levels 0 through $h - 1$ are full and hold $1 + 2 + \ldots + 2^{h-1} = 2^h - 1$ nodes. The last level has at least 1 and at most 2^h nodes. Hence $2^h \leq n \leq 2^{h+1} - 1 < 2^{h+1}$. Taking logarithms gives $h \leq \lg n < h + 1$, that is, $h = \lfloor \lg n \rfloor$. The number of levels is $h + 1$. For the second equality, the inequality $2^h < n + 1 \leq 2^{h+1}$ gives $\lceil \lg(n+1) \rceil = h + 1$. A path from the root to a node on the bottom level has h edges. ■

In the array representation this result has a simple counterpart. The level of the node with index i is $\lfloor \lg i \rfloor$, and every `i /= 2` decreases the level by one. To get from index n down to 1 takes exactly $\lfloor \lg n \rfloor$ halvings. For example, for $n = 8$, $8 \to 4 \to 2 \to 1$ is three halvings, and $\lfloor \lg 8 \rfloor = 3$. The insertion of 109 made exactly that many shifts.

Case	Example	Shifts	Cost
Best	inserting 3	0	$\Omega(1)$
Worst	inserting 109	as many as the height: $\lfloor \lg n \rfloor$	$O(\lg n)$
General	—	$0 \ldots \lfloor \lg n \rfloor$	$O(\lg n)$

Table 13.5: The cost of insertion.

Table 13.5 collects the result. The time complexity of insertion is $O(\lg n)$; its best case is $\Omega(1)$. Here O denotes an upper bound and Ω a lower bound. When the best and worst cases of an operation are of the same order, the cost is written with Θ, a tight bound. In insertion the two extremes differ, so a single Θ cannot be written; the worst case, however, is $\Theta(\lg n)$. The extra memory is $O(1)$: only the variable `i` and the parameter `item` are used.

These numbers mean a great deal at scale. In a heap of one million items, $\lfloor \lg 10^6 \rfloor = 19$; even the worst insertion makes 19 shifts. For a billion items the number is 29. When the number of items grows a thousandfold, the work grows by only 10 shifts.

For values that arrive in random order, insertion often comes nowhere near the worst case. An experiment inserted one million random values one by one into an empty heap and counted 1,279,942 shifts in total, an average of 1.3 shifts per insertion. When the same numbers were inserted in increasing order, so that each new value is larger than all earlier ones, the total was 17,951,445 shifts. This is the worst case, in which every insertion climbs to the root, and the total equals *⌊lg 1⌋ + ⌊lg 2⌋ + ... + ⌊lg n⌋*. The reason a random value most likely stays at the leaf level is simple: in a complete binary tree about half of the nodes are leaves and a quarter are one level above the leaves.

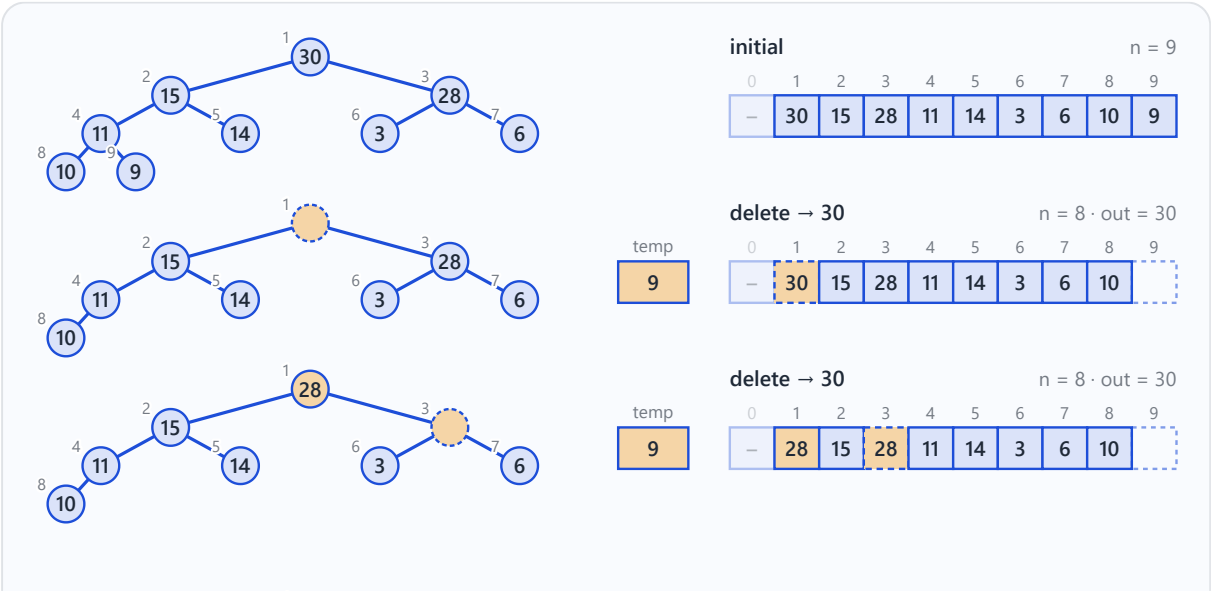
13.4 Deletion

Removing an item from a max heap is the main job of a priority queue. The item to remove is known: the largest item at the root. Reading it is easy. The hard part is turning the remaining structure back into a heap after the item is removed. As with insertion, deletion first repairs the shape and then the order; this time, though, the repair works from the top down.

13.4.1 Remove the root, place the last item

When the root is removed, a hole appears at the top of the tree. If the tree is to stay complete, the position that must be vacated when the item count drops by one is not the root but the last position, *n*. So the item in the last position is taken out of its place and a home is sought for it. The search starts from the hole at the root.

The last item is most likely a small value, because it is a leaf. If it is not larger than both children of the root, it cannot be put there. At each step three values are examined: the two children of the hole and the item to be placed. The largest of the three goes into the hole. If the largest is the item to be placed, its home has been found. Otherwise the larger child moves up one level into the hole, and the hole moves down to that child's old position. This downward movement is called **sifting down** (down-heap). The home is also found when the hole has no children left: the hole is a leaf.



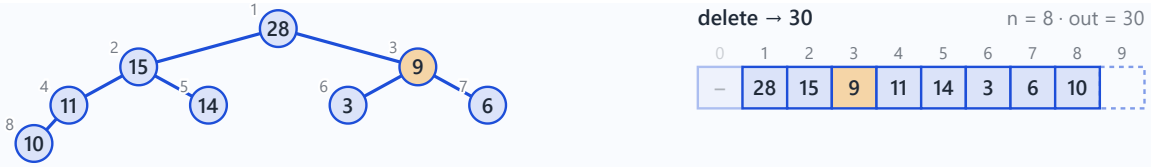


Figure 13.10: Removing the root. From top to bottom: the start; removing 30 and taking out the last item, 9; moving the larger child, 28, up; 9 settling at position 3.

Figure 13.10 shows this on the heap *30, 15, 28, 11, 14, 3, 6, 10, 9*. The first item to remove is the 30 at the root. The last item, 9, is taken out of position 9; the item count drops to 8. The children of the root are 15 and 28; the larger is 28. Since *28 > 9*, 28 moves up to the root and the hole moves down to position 3. The children of position 3 are the 3 at position 6 and the 6 at position 7; the larger is 6. Since *9 ≥ 6*, 9 settles in the hole at position 3. The new heap is *28, 15, 9, 11, 14, 3, 6, 10*.

Choosing the larger child is not arbitrary. The item that moves into the hole becomes the parent of the other child. If the smaller child moved up, the larger child would become its child and the heap property would be broken at once. When the larger child moves up, it is not smaller than its smaller sibling or than its own old children; the heap property is preserved.

13.4.2 `delete_max_heap`

The slides give a corrected version of the deletion routine from the textbook:

```
bool delete_max_heap(element *out) {
  if (HEAP_EMPTY(n)) return false;
  *out = heap[1];          /* root is returned */
  element temp = heap[n--];  /* find its slot */
  int i = 1, j = 2;
  while (j <= n) {
    if (j < n && heap[j].x < heap[j + 1].x) j++;
    if (temp.x >= heap[j].x) break;
    heap[i] = heap[j];      /* move child up */
    i = j; j *= 2;
  }
  heap[i] = temp;
  return true;
}
```

The routine first checks whether the heap is empty. If so, it returns `false` and touches nothing. Otherwise the item at the root is copied to `*out`. `out` is a pointer to the caller's variable; the assignment `*out = heap[1]` writes into the caller's variable. By definition, the first item to be removed is the first item of the array.

The line `element temp = heap[n--];` copies the item in the last position into `temp` and decrements the item count. `n--` is the postfix operator: the expression first yields the old value of *n*, and then *n* decreases. So `heap[9]` is read and *n* becomes 8. From this moment on, `heap[9]` no longer belongs to the heap. The last item has been taken out so that the complete tree stays intact, and a home will now be sought for it starting from the root.

`i` marks the candidate position of `temp`, that is, the hole; at first it is the root. `j` marks the children of `i`; at first it is 2, the left child. The loop runs as long as `j <= n`, that is, as long as `i` has at least one child. The first statement of the loop chooses the larger child: if there is a right child and it is larger than the left one, `j` is incremented. The second statement compares `temp` with the larger child. If `temp` is not smaller than it, the home has been found and the loop ends with `break`. Otherwise the larger child moves into the hole: `heap[i] = heap[j]`. The last statement moves the hole down to the moved child's old position, `i = j`, and takes `j` to its left child, `j *= 2`. When the loop exits, `temp` is written into the hole and `true` is returned.

The job of the test `j < n` is important. `j` marks the left child; the right child is at `j + 1`. If `j == n`, the left child is the last item of the heap and there is no right child. In that case `heap[j + 1]` does not belong to the heap. The condition `j < n` guarantees that the right child exists before `heap[j+1]` is read. Because the `&&` operator is evaluated with short-circuiting, the second comparison is not made at all if the condition is false.

The slides call this version “corrected”. The textbook version returns the item as the return value and writes `return 0;` for an empty heap. If the return type is a structure such as `element`, this line does not compile, because an integer cannot be converted to a structure. This line was tried in a test file, and the compiler stopped with the error “returning 'int' from a function with incompatible result type 'element'”. The suggested fix, as in the interface, is to make the return value a `bool` that reports success or failure and to return the removed item through a pointer parameter. Then there is no need to invent a meaningless item for an empty heap.

The routine is called like this:

```
element e;
if (delete_max_heap(&e))
  printf("removed: %d\n", e.x);
else
  printf("heap is empty\n");
```

`&e` is the address of the variable `e`. The assignment `*out = heap[1]` inside the routine writes to this address, that is, to `e`. In C, function parameters are passed by value. For a function to modify the caller's variable, it must receive the address of that variable. The pointer parameter provides this here.

13.4.3 Code trace: the first deletion

Table 13.6 traces the first call of `delete_max_heap` on the heap *30, 15, 28, 11, 14, 3, 6, 10, 9* line by line. The line numbers follow the trace slide in the slides: line 2 is the empty check, line 3 the copy of the root, line 4 the removal of the last item, line 6 the loop condition, line 7 the choice of the larger child, line 8 the comparison, line 9 the move, line 10 the advance, and line 12 the final write.

Line	`n`	`i`	`j`	Condition / action	`heap[1..9]`
2	9	—	—	`HEAP_EMPTY(9)` is false	30, 15, 28, 11, 14, 3, 6, 10, 9
3	9	—	—	`*out = 30`	_, 15, 28, 11, 14, 3, 6, 10, 9
4	8	—	—	`temp = 9`, $n = 8$	_, 15, 28, 11, 14, 3, 6, 10, (9)
5	8	1	2	hole at the root, `j` at the left child	_, 15, 28, 11, 14, 3, 6, 10, (9)
6	8	1	2	$2 \le 8$: enter the loop	_, 15, 28, 11, 14, 3, 6, 10, (9)
7	8	1	3	$15 < 28$: `j++`	_, 15, 28, 11, 14, 3, 6, 10, (9)
8	8	1	3	$9 \ge 28$? No	_, 15, 28, 11, 14, 3, 6, 10, (9)
9	8	1	3	`heap[1] = heap[3]`	28, 15, _, 11, 14, 3, 6, 10, (9)
10	8	3	6	`i = j; j *= 2`	28, 15, _, 11, 14, 3, 6, 10, (9)
6, 7	8	3	7	$6 \le 8$; $3 < 6$: `j++`	28, 15, _, 11, 14, 3, 6, 10, (9)
8	8	3	7	$9 \ge 6$? Yes: `break`	28, 15, _, 11, 14, 3, 6, 10, (9)
12	8	3	7	`heap[3] = temp`	28, 15, 9, 11, 14, 3, 6, 10, (9)
13	8	3	7	`return true`; the caller receives 30	28, 15, 9, 11, 14, 3, 6, 10

Table 13.6: Trace of the first `delete_max_heap`. The array column shows `heap[1..9]`, and `_` marks the hole; the cell beyond n is in parentheses.

The loop was entered twice, made one move, and ended with `break` in the second round. 9 settled at position 3 without sinking to the bottom. This is an example of “landing in the middle”: the last item found its place halfway along the path from the root to a leaf. A total of four comparisons were made: in each round one to compare the children and one to compare `temp` with the larger child. Two comparisons per round are why sifting down is slightly more expensive than the insertion loop.

The parenthesized 9 in the last column of the table shows a subtlety. After `n--` the cell 9 is not erased; it still holds a copy of 9. But since $n = 8$, none of the routines looks at that cell as part of the heap. At the next insertion, a new item will be written into that cell. In a C array, “deleting” does not empty a cell; it moves the boundary.

13.4.4 Code trace: the second deletion

At the second call the heap is *28, 15, 9, 11, 14, 3, 6, 10* with $n = 8$. This time the last item, 10, will sink all the way to a leaf. Table 13.7 summarizes the rounds of the loop.

Round	`i`	Children	Larger child (`j`)	Comparison	`heap[1..7]` (end of round)
1	1	15, 9	15 (2)	$10 \ge 15$? No: move	15, _, 9, 11, 14, 3, 6

Round	`i`	Children	Larger child (`j`)	Comparison	`heap[1..7]` (end of round)
2	2	11, 14	14 (5)	*10 ≥ 14*? No: move	15, 14, 9, 11, _, 3, 6
3	5	none (*j = 10 > 7*)	—	the loop is not entered	15, 14, 9, 11, _, 3, 6
end	5	—	—	`heap[5] = temp`	15, 14, 9, 11, 10, 3, 6

Table 13.7: Trace of the second `delete_max_heap` (`*out = 28` , `temp = 10` , *n = 7*).

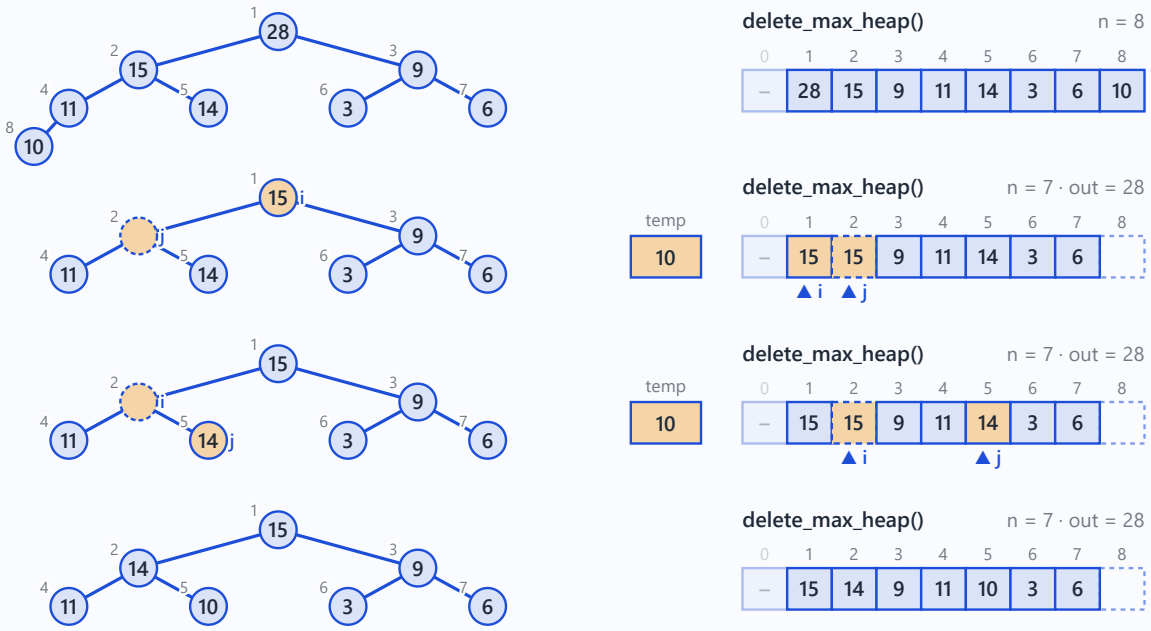


Figure 13.11: Four moments during the second deletion. From top to bottom: the start; moving 15 to the root; moving 14 to position 2; writing 10 at position 5, a leaf.

In the first round the children of the root are 15 and 9. Since the left child is not smaller than the right one, `j` stays at 2. Since *10 < 15*, 15 moves up to the root and the hole moves down to 2. In the second round the children of 2 are 11 and 14; the right one is larger and `j` becomes 5. Since *10 < 14*, 14 moves up and the hole moves down to 5. The left child of 5 would be at position 10, but *10 > n = 7*, so position 5 is a leaf. The loop condition becomes false and 10 is written into the hole. The caller receives 28; the new heap is *15, 14, 9, 11, 10, 3, 6*.

This call is the worst case of deletion: the last item sank from the root all the way to a leaf. In a heap with 8 items, the leaf level is level 2 or 3. 10 came to rest in a leaf on level 2 of the 7-item heap, having sunk two levels. The two deletions together show the two behaviors of deletion: the last item may land in the middle of the path or sink all the way to the bottom.

13.4.5 Deleting until the heap is empty

What happens if we keep removing items from the heap until it is empty? Table 13.8 shows the remaining seven removals. In each row, the removed item is the largest item, which is at the root at that moment; the last item settles into its new place by sifting down.

Removed	Last item	Position where it settled	Heap (after)
15	6	4	14, 11, 9, 6, 10, 3
14	3	5	11, 10, 9, 6, 3
11	3	4	10, 6, 9, 3
10	3	3	9, 6, 3
9	3	2	6, 3
6	3	1	3
3	3	— (it was the last item)	empty

Table 13.8: Successive removals from the heap *15, 14, 9, 11, 10, 3, 6*.
The outputs began with 30 and 28.

The output order is *30, 28, 15, 14, 11, 10, 9, 6, 3*: the items came out in decreasing order. This is no coincidence. Each removal yields the largest of the remaining items, and removing the largest items one after another produces a decreasing sequence. One detail of the table also stands out. The value 3 was the "last item" five times in a row, and each time it was placed again by a journey from the root. When 6 is removed in the sixth removal, only 3 remains in the heap. In the seventh removal 3 is both the root and the last item; `temp = heap[n--]` takes it, n becomes 0, the loop is not entered, and 3 is written into the unused cell 1. The heap is empty, and an eighth call returns `false`.

This observation is the idea of an important algorithm. If n items are put into a heap and the root is then removed n times, the items come out sorted. This is the idea of **heap sort**. Since each removal takes $O(\lg n)$ time, the sort takes $O(n \lg n)$ time. A version of heap sort that uses no extra array also appears in these notes.

13.4.6 The cost of deletion

The cost of deletion too is determined by the number of iterations of the loop. Each iteration moves the hole down one level and makes at most two comparisons and one move. The hole starts at the root and must stop at the bottom level at the latest. By Theorem 13.2, at most $\lfloor \lg n \rfloor$ iterations are made. The worst case of deletion is therefore $O(\lg n)$; the number of comparisons is at most $2\lfloor \lg n \rfloor$.

The best case of deletion is also $\Omega(1)$. If the last item is not smaller than both children of the root, it settles in the first round. This happens, for example, when two items remain in the heap: when the root is removed, n drops to 1, the loop is never entered because $j = 2 > n$, and the last item is written directly at the root. In a heap with many items of equal priority, the last item often settles immediately too. But in a large heap with distinct values, the last item comes from a leaf and is usually small, so it sinks down to near the leaves. For this reason, unlike insertion, deletion runs close

to the worst case on average even for random data. In a heap built from one million random values, the first thousand deletions made 18.93 rounds on average; $\lfloor \lg 10^6 \rfloor = 19$. The extra memory is again *O(1)*: `temp`, `i` and `j`.

> **Common mistake**
>
> In deletion, moving the larger child of the root up to the root and carrying the hole downward, but not filling the hole with the last item in the end. This method leaves the hole at an arbitrary position on the leaf level. If the hole is not in the last position, a gap appears in the array and the tree stops being complete. In deletion the shape is preserved first: the position removed is always *n*, and the item whose home is sought is the one that comes from there.

13.4.7 Two faulty versions

The deletion routine looks small, but details in two of its lines determine its correctness. Why these details are needed can be seen from two versions that break them.

The first child instead of the larger child. If the line `if (j < n && heap[j].x < heap[j + 1].x) j++;` is removed from the loop, the left child is chosen in every round. With this version, 30 was removed from the heap *30, 15, 28, 11, 14, 3, 6*. The last item is 6. The left child, 15, moves up to the root because *6 < 15*; the hole moves down to 2, where the left child 11 moves up, and 6 is written at position 4. The result is *15, 11, 28, 6, 14, 3*. The 15 at the root is smaller than its right child 28; the heap property is broken. The next removal returns 15, not 28, as the largest item. The program does not crash; it just returns the wrong item.

Without the `j < n` *test.* If this test is skipped, `heap[n + 1]` is read when `j == n`. This cell is outside the heap. In `delete_max_heap` there is an interesting situation: after `n--`, `heap[n + 1]` holds the copy of `temp` itself, the item whose home is being sought. If that copy is selected, the comparison becomes `temp.x >= temp.x` and the loop stops at the right place. For this reason, in this routine the bug does not change the result; even in half a million random trials of mixed insertions and removals no wrong result was seen. But that is an accident and does not show that the code is correct. When the same loop is used in bottom-up heap construction, `a[n + 1]` may lie outside the array. In an array of 6 items, when this test was removed, the value 99 just after the array entered the heap and a 3 in the array was lost. The `j < n` test must therefore be written in every sift-down loop.

13.5 Improving the implementation

The insertion and deletion routines work correctly, but they depend on a global array. This section improves the routines in three directions: moving the heap into a structure, writing the min heap, and handling items whose priority changes over time.

13.5.1 A structure instead of a global array

With a global array, a program can have only one heap. An operating system, however, keeps many queues: a separate ready queue for each processor, and another queue for timers. The capacity is also fixed at compile time. To remove these drawbacks, the slides gather the array and the item count in one structure and allocate the array at run time:

```
typedef struct { element *a; int n, capacity; } Heap;

Heap *heap_create(int capacity) {
  Heap *h = malloc(sizeof *h);
  if (h == NULL) return NULL;
  h->a = malloc((capacity + 1) * sizeof *h->a);
  if (h->a == NULL) { free(h); return NULL; }
  h->n = 0; h->capacity = capacity;
  return h;
}
void heap_destroy(Heap *h) { if (h) { free(h->a); free(h); } }
```

The `Heap` structure has three fields. `a` is the address of the array allocated on the heap memory. `n` holds the item count, and `capacity` holds the maximum number of items it can take. Since every heap is a separate `Heap`, a program can have as many heaps as it wants. The routines take the heap they work on as a parameter.

`heap_create` allocates two separate blocks. The first is the structure itself: `malloc(sizeof *h)`. Here `sizeof *h` is the size of the type that `h` points to, that is, `Heap`. Not repeating the type name keeps the line correct if the type changes later. The second block is the array. Because cell 0 is unused, `capacity + 1` cells are allocated. The result of both allocations is checked. If the second allocation fails, it is not enough to return `NULL`; the structure allocated earlier must also be given back with `free(h)`. Otherwise that block would remain as a piece of memory nobody can point to; that is, it would leak.

`heap_destroy` releases the blocks in the reverse order of allocation. The array is released first and the structure second, because the address of the array is inside the structure; if the structure were released first, `h->a` could not be read. The `if (h)` check makes a call with a `NULL` pointer harmless. This is consistent with the fact that `free(NULL)` does nothing.

The slides do not give the insertion and removal routines of the structure version; the suggestion is to report success with `true` / `false` and to pass the item through a pointer. The following two routines apply this suggestion. Their bodies are the same as the global versions; only `h->a` is written instead of `heap` and `h->n` instead of `n`:

```c
bool heap_insert(Heap *h, element item) {
  if (h->n == h->capacity) return false;   /* full */
  int i = ++h->n;
  while (i != 1 && item.x > h->a[i / 2].x) {
    h->a[i] = h->a[i / 2];
    i /= 2;
  }
  h->a[i] = item;
  return true;
}

bool heap_pop(Heap *h, element *out) {
  if (h->n == 0) return false;             /* empty */
  *out = h->a[1];
  element temp = h->a[h->n--];
  int i = 1, j = 2;
  while (j <= h->n) {
    if (j < h->n && h->a[j].x < h->a[j + 1].x) j++;
    if (temp.x >= h->a[j].x) break;
    h->a[i] = h->a[j];
    i = j; j *= 2;
  }
  h->a[i] = temp;
  return true;
}
```

In the insertion routine, a full heap is no longer reported with an error message but with a return value of `false`; the caller decides what to do. The loop condition has also become a little shorter: the conditions `i != 1` and “the parent is smaller” are combined in a single `while`. Because `&&` is evaluated with short-circuiting, `h->a[0]` is never read when `i == 1`. The two forms do the same job.

Figure 13.12 draws this structure as it appears in memory. Only the pointer `h` lives on the program stack. The structure and the array are in heap memory; the `a` field of the structure holds the starting address of the array.

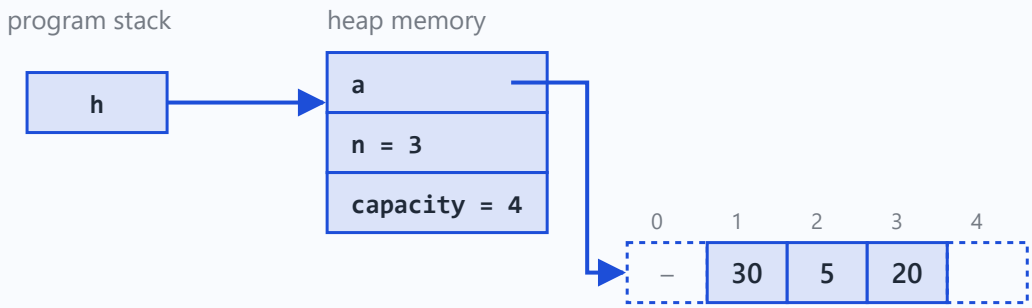


Figure 13.12: Memory view of a heap created with `heap_create(4)` into which 20, 5 and 30 have been

inserted. The array has *capacity + 1 = 5* cells; cell 0 is unused and cell 4 is still empty.

This version was tested with a counting `malloc` / `free` wrapper. The wrapper consists of two small functions that increment a counter on every successful allocation and every release; they are substituted for `malloc` and `free` with macros. The test began with `heap_create(4)` and tried to insert the values 20, 5, 30, 10, 40. The first four insertions returned `true` and the fifth returned `false`, because the capacity is 4. Then items were removed until the heap was empty, and *30, 20, 10, 5* came out. After `heap_destroy`, the counters showed 2 allocations and 2 releases; there is no leak. The call `heap_destroy(NULL)` also passed without trouble. With this compiler configuration, `sizeof(Heap)` is 16 bytes: an 8-byte pointer and two 4-byte `int`s.

The capacity of the array can also be made unbounded by doubling it with `realloc` when it fills up. Then insertion never returns `false`; the single insertion that triggers the growth takes $\Theta(n)$, but the amortized cost does not change. The total copying cost of doubling does not exceed $2n$ over n insertions.

13.5.2 Min heap

The photocopier and airport examples would call for a min priority queue. The routines of a min heap differ from those of a max heap in only two respects: the name of the routine and the direction of the comparisons. The slides give the insertion routine:

```
void insert_min_heap(element item) {
  /* ... same full check ... */
  int i = ++n;
  while (i != 1) {
    if (item.x >= heap[i / 2].x) break; /* >= ! */
    heap[i] = heap[i / 2];
    i /= 2;
  }
  heap[i] = item;
}
```

The only change is in the `break` condition: `item.x >= …` instead of `item.x <= …`. As long as the new item is smaller than its parent, the parent moves down; if the new item is not smaller than its parent, its place has been found. The comment on the second line of the code says that the full check is the same as in the max version; it is omitted for brevity. In the test program it was compiled with this check added.

In the removal routine, two places change: the smaller child is chosen instead of the larger one, and the stopping condition is reversed.

```
bool delete_min_heap(element *out) {
  if (HEAP_EMPTY(n)) return false;
  *out = heap[1];
  element temp = heap[n--];
  int i = 1, j = 2;
  while (j <= n) {
    if (j < n && heap[j].x > heap[j + 1].x) j++;  /* smaller child */
    if (temp.x <= heap[j].x) break;               /* <= ! */
    heap[i] = heap[j];
    i = j; j *= 2;
  }
  heap[i] = temp;
  return true;
}
```

Place	Max	Min
insert: stopping condition	`item.x <= parent`	`item.x >= parent`
delete: choice of child	the larger: `j++` if `[j] < [j+1]`	the smaller: `j++` if `[j] > [j+1]`
delete: stopping condition	`temp.x >= [j]`	`temp.x <= [j]`

Table 13.9: Differences between the max heap and min heap routines.

If the name of the routine is changed and the less-than and greater-than signs are reversed, the min heap is written. Table 13.9 collects the three lines that change. These two routines were tried with the values 20, 5, 30, 10, 40. After the insertions the array became, in turn, *20*; *5, 20*; *5, 20, 30*; *5, 10, 30, 20*; *5, 10, 30, 20, 40*. The removal calls returned *5, 10, 20, 30, 40*, that is, increasing order.

Another way to reverse the comparisons is to leave the max heap untouched and multiply the priorities by minus one. The largest $-x$ is the smallest x. This trick is used in practice, but it is limited to cases where the key is a number, and it risks overflow at boundary values such as `INT_MIN`. A more general solution is to move the comparison into a separate function: the routines call that function, and which kind of heap it is gets determined by it. The `qsort` of the standard library likewise receives the sort direction through such a comparison function.

13.5.3 Items whose priority changes

Let us return to the airport example. The fuel of an aircraft is not measured once, at the moment it first contacts the tower, and then forgotten. Every aircraft burns fuel at a different rate. One aircraft may burn 10 gallons (37.85 liters) in a second, while another might fly an hour on 10 gallons. Priorities therefore change over time. In the system an interrupt arrives every second and the fuel levels are updated. Capturing the problem well requires including this change in the model.

When the priority of an item changes, the heap property can break only around that item. In a max heap, an item whose priority increases may become larger than its parent; it is sifted up. An item whose priority decreases may become smaller than its children; it is sifted down. Both operations

work along a single root-to-leaf path and take *O(lg n)*. The slides give the sift-up version:

```
/* priority of the item at k rose: sift it up */
void increase_key(int k, int key) {
  element e = heap[k];
  e.x = key;
  while (k != 1 && e.x > heap[k / 2].x) {
    heap[k] = heap[k / 2];
    k /= 2;
  }
  heap[k] = e;
}
```

The routine copies the item at position *k* into `e` and sets its priority to the new value. Then it runs the same loop as the insertion routine, except that the starting position is *k*, not *n + 1*. The condition `k != 1` stops at the root; the condition `e.x > heap[k / 2].x` continues as long as the parent is smaller. At the end of the loop the item is written to its new place. The routine assumes that the new value is larger than the old one. If it is called with a smaller value, the loop does not run at all and the item just shrinks in place; this may break the heap property downward.

When a priority decreases, sifting down is needed. This version is not in the slides; it is obtained by starting the loop of the deletion routine at position *k*:

```
/* priority of the item at k fell: sift it down */
void decrease_key(int k, int key) {
  element e = heap[k];
  e.x = key;
  int j = 2 * k;
  while (j <= n) {
    if (j < n && heap[j].x < heap[j + 1].x) j++;
    if (e.x >= heap[j].x) break;
    heap[k] = heap[j];
    k = j; j *= 2;
  }
  heap[k] = e;
}
```

Example 13.3

Let the heap be *30, 15, 28, 11, 14, 3, 6, 10, 9*. The priority of the 9 at position 9 is raised to 31: `increase_key(9, 31)`. Its parent is the 11 at position 4, and *31 > 11*; 11 moves down. Then the 15 at position 2 and the 30 at the root move down too. 31 is written at the root. The result is *31, 30, 28, 15, 14, 3, 6, 10, 11*; the path is 9 → 4 → 2 → 1, three shifts.

In the same initial heap, the priority of the root is lowered to 5: `decrease_key(1, 5)`. The children of the root are 15 and 28; 28 moves up. The children of position 3 are 3 and 6; 6 moves up. Position 7 is a leaf; 5 is written there. The result is *28, 15, 6, 11, 14, 3, 5, 10, 9*. Both results were checked with `is_heap` and found valid.

The hard part of these routines is not the code but knowing *k*. The tower knows the name of the aircraft whose fuel changed; it does not know its position in the heap. Searching for an item in a heap takes *O(n)*, because a heap is not a search structure. If a search is made for every update, the gain of the *O(lg n)* sift is lost. The solution is to keep the position of every item in the heap in a separate **index table**, for example `pos[plane_id]`. Every time a heap routine moves an item, it updates this table as well. If an item is moved with `heap[i] = heap[j]`, then `pos[heap[i].id] = i` is written right after. This way the position is found in *O(1)* and changing a priority really takes *O(lg n)*. The **decrease-key** operation of graph algorithms is done the same way in a min heap.

13.6 Comparison with other structures

Both operations of the heap take *O(lg n)*. Is that a good result? Could a simpler structure, an array or a list, not do the same job more easily? This section compares five implementations, first by their theoretical costs, then in numbers, and finally in a real measurement.

13.6.1 The cost of five implementations

#	Structure	Insert	Delete (largest)	Why
1	Unsorted array	*Θ(1)*	*Θ(n)*	Insert at the end; finding the largest means scanning everything.
2	Unsorted list	*Θ(1)*	*Θ(n)*	Insert at the head; again every node is traversed.
3	Sorted array (ascending)	*O(n)*	*Θ(1)*	`binsearch` *O(lg n)* + *n* shifts; the largest is at the end.
4	Sorted linked list (descending)	*O(n)*	*Θ(1)*	Find the place; delete from the head and update the head.
5	Max heap	*O(lg n)*	*O(lg n)*	Only one root-to-leaf path.

Table 13.10: Five implementations of a max priority queue.

Table 13.10 gives the comparison from the slides. The rows can be taken one at a time.

Unsorted array. The new item is written at the end of the array: one assignment, *Θ(1)*. To remove the largest item, it must first be found. Because the array is unsorted, all items are examined; this is always $n - 1$ comparisons, and *Θ(n)*. When the item found is removed, the last item is moved into the gap; no shifting is needed. Removal therefore takes exactly linear time.

Unsorted list. In a linked list, the new item is added at the head: one node is allocated and two pointers are updated, $\Theta(1)$. Finding the largest again requires traversing all nodes: $\Theta(n)$.

Sorted array. If the array is kept in ascending order, the largest item is at the end and removal is $\Theta(1)$: n is decremented. Insertion, however, is expensive. The place of the new item is found by binary search (`binsearch`) in $O(\lg n)$, but all items larger than it must be shifted one place to the right. If the new item is the smallest, n shifts are made. If the new item is the largest, no shift is made at all; so insertion is $O(n)$, not $\Theta(n)$. If binary search is not used and the place is found by linear search from the start, the situation changes. Finding the place examines k items, and then $n - k$ items are shifted; the total is always about n. Then insertion is $\Theta(n)$, not just $O(n)$.

Sorted linked list. If the list is kept in descending order, the largest item is at the head. Removal means taking out the head node and updating the head: $\Theta(1)$. In insertion, the place of the new item is found by traversing from the head; no shifting is needed, but binary search cannot be done on a linked list. If the new item is the largest, it is added at the head at once; if it is the smallest, the whole list is traversed. Insertion is therefore $O(n)$.

Max heap. Both operations work along a single root-to-leaf path: $O(\lg n)$.

The pattern of the table is clear. Each simple structure makes one of the operations very cheap, but pays for it in the other. Unsorted structures keep insertion cheap because they maintain no order; the price is searching for the largest again every time. Sorted structures keep removal cheap because they maintain a full order; the price is rebuilding that order at every insertion. The heap stands between the two. It does not sort completely; it maintains only a weak order, "every parent is larger than its children". This order is enough to keep the largest at the root, and repairing it requires work along only one path.

Another idea that comes to mind is to keep the largest value in a separate variable `max`. In addition to the unsorted array, this variable is updated at every insertion, and looking at the largest becomes $O(1)$. But what happens when the largest item is removed, that is, when that aircraft lands? To find the new largest, the whole array must be scanned again. To avoid this one would have to keep the second largest, and for that the third as well; this is no different from using a sorted array.

13.6.2 In numbers: n = 2,000,000

The cost table becomes concrete with a number. Let $n = 2{,}000{,}000$ items, and let every basic operation take one second. This assumption is exaggerated, but it makes the ratios easy to see. $\lg 2{,}000{,}000 \approx 20.9$; it is rounded to 21. That is also why 2 million was chosen: its logarithm is easy to compute, since $2^{21} = 2{,}097{,}152$.

Structure	Insert	Delete
1 · unsorted array	1	2,000,000
2 · unsorted list	1	2,000,000
3 · sorted array	2,000,000	1

Structure	Insert	Delete
4 · sorted list	2,000,000	1
5 · max heap	≈ 21	≈ 21

Table 13.11: Number of steps per operation in the worst case for $n = 2{,}000{,}000$ (one step = one second, by assumption).

Table 13.11 compares the five approaches by analysis alone, without writing any code. In the first four structures, one of the operations takes 2 million seconds, about 23 days. In the heap both operations take about 21 seconds. Since a queue performs insertions and removals in roughly equal numbers, looking at the sum of one insertion and one removal is a fair measure. For the simple structures this sum is about 2 million steps; for the heap it is 42 steps. The heap is about 48,000 times faster. By way of analogy: running a program built on a heap forty thousand times takes less time than running it once with one of the other structures.

A real instruction takes not a second but a small fraction of a microsecond. One may therefore raise this objection: "An aircraft arrives at the airport every half hour; even with a queue of a thousand aircraft, the worst structure runs in the blink of an eye. Who will notice that the algorithm is bad?" For a small problem this may be true. But when the same code is part of an operating system, it is called thousands of times a second and the difference spreads through the whole system. In work that demands expertise, a solution that collapses when the workload grows has no academic or scientific value.

13.6.3 Measurement

The theoretical comparison was also tested with a measurement. Three implementations were written in C and compiled with `-O2`: an unsorted array, a sorted array with linear shifting, and a max heap. Into each, n random values were inserted, and then the largest was removed n times. The sums of the values returned by the three structures came out equal in every run; that is, all three returned the same items. Table 13.12 gives the ranges of values over three runs. The values vary from machine to machine; what matters is the shape of the growth.

n	Unsorted array	Sorted array	Max heap
10,000	0.12–0.16	0.011–0.015	≈ 0.001
50,000	3.3–3.6	0.57–0.77	0.008–0.019
100,000	16.0–17.1	2.3–2.9	0.017–0.027

Table 13.12: Time for n insertions + n removals (seconds; three runs; varies by machine).

When n grew tenfold, the time of the unsorted and sorted arrays grew about a hundredfold. This is quadratic growth: when each of n operations takes $\Theta(n)$, the total is $\Theta(n^2)$. The heap's time grows roughly as $n \lg n$. At one hundred thousand items the heap is about 700 times faster than the unsorted array and about 100 times faster than the sorted array. That the sorted array comes out

faster than the unsorted array is instructive as well. Both do $\Theta(n^2)$ total work. But shifting is a very simple loop over consecutive memory cells; between two methods of the same growth class, the constant factor can differ this much. The asymptotic class, however, is stronger than the constant factor: as n grows, the gap to the heap keeps widening.

13.6.4 Advanced heaps

The structure described in these notes is the **binary heap**. There are other, more advanced kinds of heaps that do some operations even faster. Researchers did not think $lg\ n$ was enough. As in the Olympic motto: *Citius! Altius! Fortius!* (Faster! Higher! Stronger!) Table 13.13 compares three structures by their min heap operations.

Operation	Binary heap	Binomial heap	Fibonacci heap*
insert	$O(lg\ n)$	$O(lg\ n)$	$O(1)$
find minimum	$O(1)$	$O(lg\ n)$	$O(1)$
delete minimum	$O(lg\ n)$	$O(lg\ n)$	$O(lg\ n)$
decrease key	$O(lg\ n)$	$O(lg\ n)$	$O(1)$
merge two heaps	$O(n)$	$O(lg\ n)$	$O(1)$

Table 13.13: Operation costs of binary, binomial and Fibonacci heaps. * Fibonacci heap times are amortized.

A **binomial heap** is not a tree but a forest made up of trees of particular sizes. Its main gain is merging: two binomial heaps can be merged in $O(lg\ n)$. In a binary heap, merging means concatenating two arrays and rebuilding the heap, which takes $O(n)$. The **Fibonacci heap** postpones the work even further. During insertion and decrease-key it does not tidy the structure; it leaves the tidying to the moment when the minimum is deleted. That is why its times are amortized: a single operation may be expensive, but when the total cost of a sequence of operations is divided by the number of operations, the values in the table result. In graph algorithms that perform many decrease-key operations, such as Dijkstra and Prim, the Fibonacci heap gives a theoretical gain. In practice, the simplicity of the binary heap and its cache-friendly layout in an array often give faster results.

13.7 Bottom-up heap construction

Sometimes the items do not arrive one by one; all of them are ready in an array from the start. Heap sort, for example, takes the array to be sorted all at once. The first way that comes to mind to turn such an array into a heap is to insert the items one by one into an empty heap. Since each insertion takes $O(lg\ n)$, this way takes $O(n\ lg\ n)$. There is a better way: fixing the array in place, from the bottom up. This method is called **bottom-up heap construction** (heapify) and takes $O(n)$.

13.7.1 `build_heap`

The starting point of the idea is the leaves. A tree consisting of a single node is already a heap; it has no children to compare with. In an array of n items, all nodes with $k > \lfloor n/2 \rfloor$ are leaves, because their left children would be at position $2k > n$. So the second half of the array, before anything has been done, consists of one-item heaps.

Now we go backward from $k = \lfloor n/2 \rfloor$ toward the root. For each k, the two subtrees of the node are already heaps, because all nodes with larger indices have been fixed earlier. In that case only the item at k may be out of place. This is exactly the situation in deletion: an item whose place is sought at the root, and two heaps. The item at k is sifted down, and the subtree rooted at k becomes a heap. When the loop reaches the root, the whole array is a heap.

```
void build_heap(element a[], int n) {
  for (int k = n / 2; k >= 1; k--) {
    element temp = a[k];
    int i = k, j = 2 * k;
    while (j <= n) {
      if (j < n && a[j].x < a[j + 1].x) j++;
      if (temp.x >= a[j].x) break;
      a[i] = a[j];
      i = j; j *= 2;
    }
    a[i] = temp;
  }
}
```

The outer loop counts backward from `k = n / 2` down to 1. The inner part is the same as the loop in `delete_max_heap`; the only difference is that the starting position is k rather than 1. In Horowitz's book this inner loop is a separate routine called `adjust`. `temp` holds the item at k; `i` marks the hole and `j` the children. As long as the larger child is larger than `temp`, it is moved up and the hole moves down. At the end `temp` is written into the hole.

The routine uses no extra array; the heap is built inside the input array. Its extra memory is therefore $O(1)$. Such algorithms are called **in-place** algorithms. The array is 1-based: the items run from `a[1]` to `a[n]` and `a[0]` is unused. The caller must supply an array of at least $n + 1$ cells.

13.7.2 Code trace: `build_heap(a, 7)`

The input is *5, 3, 8, 9, 1, 12, 7* and is not yet a heap; the 5 at the root is smaller than its children. $n = 7$, $\lfloor n/2 \rfloor = 3$. Positions 4, 5, 6 and 7 are leaves. The outer loop runs for 3, 2 and 1. Table 13.14 summarizes the inner loop for each k.

`k`	`temp`	**Steps**	**Position where it settled**	`a[1..7]` **(after)**
—	—	input	—	5, 3, 8, 9, 1, 12, 7
3	8	children 12 and 7, the larger is `a[6] = 12`; *8 < 12*: 12 moves up; *j = 12 > n*	6	5, 3, 12, 9, 1, 8, 7
2	3	children 9 and 1, the larger is `a[4] = 9`; *3 < 9*: 9 moves up; *j = 8 > n*	4	5, 9, 12, 3, 1, 8, 7
1	5	children 9 and 12, the larger is `a[3] = 12`; *5 < 12*: 12 moves up. Children 8 and 7, the larger is `a[6] = 8`; *5 < 8*: 8 moves up; *j = 12 > n*	6	12, 9, 8, 3, 1, 5, 7

Table 13.14: Trace of `build_heap(a, 7)`.

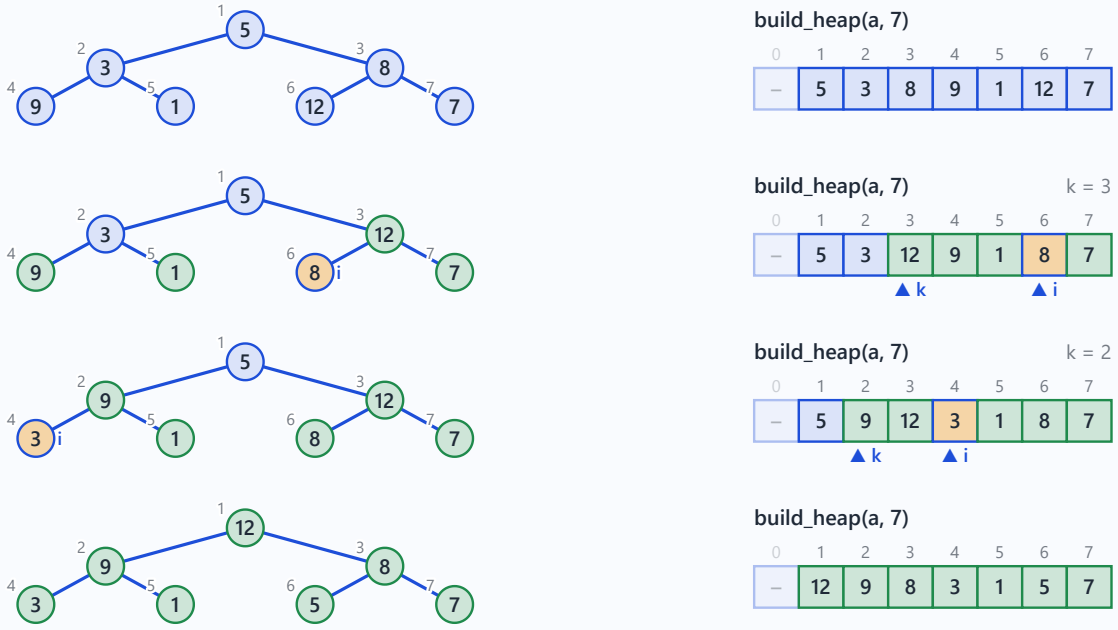


Figure 13.13: Four moments of `build_heap`. From top to bottom: the input; the end of the steps *k = 3*, *k = 2* and *k = 1*. Green nodes are roots whose own subtrees are already heaps.

At *k = 3*, 8 looks as if it swapped places with 12; in fact 12 was moved up and 8 was written into the hole. The subtree rooted at 3 is now *12, 8, 7*. At *k = 2*, similarly, 9 moves up and 3 sinks to a leaf. After both subtrees are heaps, the root is fixed at *k = 1*. This time the path of 5 is two levels: first 12, then 8 moves up, and 5 settles at position 6. The outer loop ends at *k = 0*. The result *12, 9, 8, 3, 1, 5, 7* is a valid max heap. In all, the routine made 4 moves and, for each *k*, one final write, that is, 3 final writes.

If the same seven values had been inserted one by one into an empty heap with `insert_max_heap`, the result would have been *12, 8, 9, 3, 1, 5, 7*. This is also a valid max heap, but it differs from what `build_heap` gives. A given set of values has more than one heap; the two methods arrive at different ones. Which one is obtained does not matter. In both the root is the largest item, and subsequent removals give the same order.

13.7.3 Why *O(n)*?

At first glance the cost of `build_heap` can be computed as follows: the outer loop runs *n/2* times, and each time the inner loop takes *O(lg n)*; the total is *O(n lg n)*. This bound is correct but loose. The key observation is that the great majority of nodes are in the lower part of the tree, and the nodes there can sink only very little.

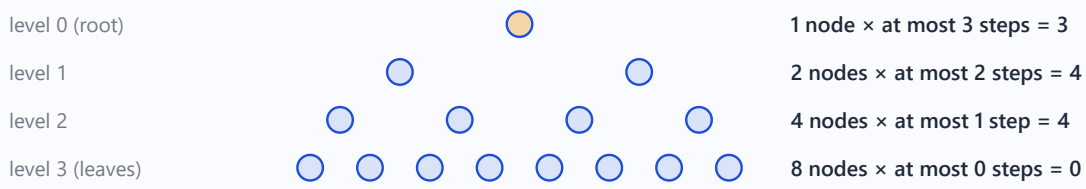


Figure 13.14: Maximum number of levels each node can sink, by level, in a complete tree with 15 nodes. The root can sink 3 levels; the leaves never sink.

A node can sink at most as many levels as there are below it. This number is called the **height** of the node; the height of a leaf is 0. About half of the nodes are leaves and never sink. A quarter are directly above the leaves and sink at most 1 level. An eighth sink at most 2 levels. Only a single node, the root, can sink $\lfloor \lg n \rfloor$ levels. Figure 13.14 does this computation for a complete tree with 15 nodes: $0 \cdot 8 + 1 \cdot 4 + 2 \cdot 2 + 3 \cdot 1 = 11$. The total number of sinking steps is at most 11, and this number is smaller than $n = 15$.

Theorem 13.3 (Cost of bottom-up construction)

`build_heap` turns an array of n items into a max heap in $O(n)$ time. The total number of sinking steps is less than n.

Proof. In a heap of n items, the number of nodes of height h is at most $\lceil n / 2^{h+1} \rceil$. Sifting down a node of height h takes at most h rounds, and each round does constant work. The total work is bounded as follows:

$$\Sigma_{h \geq 0} \lceil n / 2^{h+1} \rceil \cdot O(h) = O(n \cdot \Sigma_{h \geq 0} h / 2^{h}) = O(2n) = O(n)$$

Here $\Sigma\, h / 2^h = 1/2 + 2/4 + 3/8 + \ldots = 2$. The value of this sum can be seen as follows. Let $S = \Sigma_{h \geq 1} h / 2^h$. Then $S / 2 = \Sigma_{h \geq 1} h / 2^{h+1} = \Sigma_{h \geq 2} (h - 1) / 2^h$. Subtracting gives $S - S/2 = 1/2 + \Sigma_{h \geq 2} 1/2^h = 1/2 + 1/2 = 1$, so $S = 2$. A more exact count is possible for a full tree: in a perfect tree with $2^m - 1$ nodes, the total number of sinking steps is at most $n - m$, that is, less than n. In the 15-node example this is $15 - 4 = 11$. ■

Why does repeated insertion not give the same cheapness? In insertion the movement is upward, and the number of levels a node can climb is its *depth*, that is, its distance from the root. Half of the nodes are on the bottom level, and each of them can climb $\lfloor \lg n \rfloor$ levels to the root. The crowded lower levels can walk the longest path; the total reaches $\Theta(n \lg n)$. In bottom-up construction the situation is reversed: only the few nodes near the top walk the long path.

The two methods were compared in a measurement. On random arrays, the numbers of comparisons and moves made by `build_heap` were counted, and the same values were also inserted one by one. Table 13.15 gives the results.

n	`build_heap` comparisons	`build_heap` moves	Insertion (random) shifts	Insertion (increasing) shifts
1,000	1,893	762	1,337	7,987
10,000	18,841	7,460	12,898	113,631
100,000	188,069	74,406	128,122	1,468,946
1,000,000	1,881,022	743,338	1,279,942	17,951,445
10,000,000	18,815,235	7,441,057	12,817,582	213,222,809

Table 13.15: Counts for bottom-up construction and for repeated insertion. The last column is the number of shifts in insertion in increasing order (the worst case).

The counts of `build_heap` grow exactly in proportion to n: about $1.88n$ comparisons and about $0.74n$ moves. The number of shifts in insertion in increasing order, in contrast, grows by more than a factor of ten when n grows tenfold; there are 8 shifts per item at a thousand items and 21 per item at ten million. This is $n \lg n$ growth. The average for insertion in random order stays at about 1.3 shifts per item, but this average is not a guarantee. `build_heap`, on the other hand, makes fewer than n moves on every input. At ten million items, the times of both methods were measured at around 0.1 seconds on this machine (`build_heap` 0.09–0.12 s, random insertion 0.13–0.15 s).

13.8 d-ary heap

In a binary heap every node has at most two children. This number does not have to be 2. A heap can also be built on a complete tree in which every node has at most d children.

Definition 13.4

A **d-ary heap** is a tree in which every node has at most d children, every level except the last is full, and the last level fills from left to right; the value of every node is not smaller than those of its children (the max version). $d = 2$ gives the binary heap.

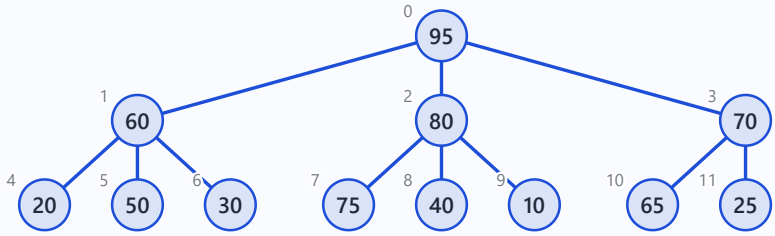


Figure 13.15: A 3-ary max heap with 12 items. Indices start at 0; the children of node *i* are *3i + 1*, *3i + 2*, *3i + 3*.

A d-ary heap is also stored in an array. In a 0-based array, the children of node *i* run from $d \cdot i + 1$ to $d \cdot i + d$, and its parent is $\lfloor (i - 1)/d \rfloor$. For $d = 2$ these formulas are the same as those of the 0-based binary heap. Figure 13.15 shows a 3-ary heap. The children of the root, 95, are 60, 80 and 70 at indices 1, 2 and 3. The children of the 80 at index 2 run from $3 \cdot 2 + 1 = 7$ to 9: 75, 40 and 10. The 70 at index 3 has only two children, at indices 10 and 11: 65 and 25. The tree is complete; the last level fills from left to right.

	Binary ($d = 2$)	**d-ary**
parent	`(i - 1) / 2`	`(i - 1) / d`
children	`2i + 1`, `2i + 2`	`d·i + 1` … `d·i + d`
height	$\approx \lg n$	$\approx \log_d n$
insert / change a key toward the root	$O(\lg n)$	$O(\log_d n)$: faster
delete the root	$O(\lg n)$	$O(d \cdot \log_d n)$: d children are compared

Table 13.16: The binary heap and the d-ary heap (0-based array).

As the number of children grows, the tree becomes shallower: the height drops from $\lg n$ to $\log_d n = \lg n / \lg d$. For example, with a million items the height of a binary heap is 19, and that of a 4-ary heap is about 10. Since sifting up makes only one comparison per level, insertion becomes faster. Sifting down, however, must find the largest of d children at every level; this takes $d - 1$ comparisons. Deleting the root therefore takes $O(d \cdot \log_d n)$ and gets slower as d grows.

Because the table in the slides speaks of a max heap, it calls the operation that requires sifting up “increase key”. In a min heap the same operation is “decrease key”. Dijkstra's shortest-path algorithm uses a min heap and performs many decrease-key operations and relatively few deletions. In dense graphs, where the number of edges is large compared with the number of vertices, this ratio is pronounced. In such cases choosing $d > 2$ pays off. A shallow tree also has the advantage of being cache-friendly: the d children of a node sit side by side in the array and often fall on the same cache line.

13.9 Applications and heap sort

A priority queue is used wherever the question “which is the next most important job?” is asked again and again. The slides list six areas.

- **Operating systems.** Scheduling of processes and interrupts, timer events. The timer that will expire soonest sits at the root of a min heap.
- **Graph algorithms.** Dijkstra's shortest-path algorithm and Prim's minimum spanning tree algorithm choose, at every step, the nearest vertex not yet processed. This choice is made with a min heap.
- **Heap sort.** Build a heap, then remove the root n times: $O(n \lg n)$, in-place sorting.
- **Event simulation.** Events are drawn from a min heap according to their timestamps; each event may add new future events.
- **The largest or smallest k items.** In a stream of data, the largest k items are tracked with a min heap of size k. If a new value is larger than the root, the root is discarded and the new value is inserted. Each step takes $O(\lg k)$ and the whole stream takes $O(n \lg k)$. For the smallest k items the same job is done with a max heap.
- **Huffman coding.** In text compression, the two trees with the smallest frequencies are merged repeatedly. Finding the two smallest at every step is a job for a min heap.

The shortest of these applications is heap sort, and it brings together all the pieces examined in the notes. We saw that repeated removal yields decreasing order. To apply this idea without an extra array, one observation suffices: when the root is removed, the heap shrinks by one item and a cell at the end of the array becomes free. The largest item that was removed can be placed in exactly that cell. The end of the array thus becomes the region where the sorted items accumulate. The following routine applies this idea:

```
/* sorts a[1..n] in ascending order; no extra array */
void heap_sort(element a[], int n) {
  build_heap(a, n);                /* O(n) */
  for (int m = n; m > 1; m--) {
    element temp = a[m];           /* last item */
    a[m] = a[1];                   /* largest to the end */
    int i = 1, j = 2;              /* fix a[1..m-1] */
    while (j <= m - 1) {
      if (j < m - 1 && a[j].x < a[j + 1].x) j++;
      if (temp.x >= a[j].x) break;
      a[i] = a[j];
      i = j; j *= 2;
    }
    a[i] = temp;
  }
}
```

The routine first turns the array into a heap with `build_heap`. Then `m`, the current size of the heap, goes down from *n* to 2. In each round the last item is taken into `temp`, the largest item at the root is written to `a[m]`, and a place for `temp` is sought within `a[1..m-1]` starting from the root. This last part is the loop of `delete_max_heap`; only the end of the heap is `m - 1`. When the loop ends, `a[2..n]` is sorted and the smallest item is in `a[1]`.

The array *5, 3, 8, 9, 1, 12, 7* was sorted by this routine into *1, 3, 5, 7, 8, 9, 12*. The routine was also tested with 3000 random arrays of 0–149 items containing duplicate values, and gave increasing order every time. The cost is the sum of *O(n)* for `build_heap` and *O(n lg n)* for the *n − 1* sift-downs: *O(n lg n)*. This time bound holds in the worst case too. The extra memory is only a few variables: *O(1)*.

13.10 Complexity summary

Operation	Time	Extra memory	Comment
look at the largest	*O(1)*	*O(1)*	`heap[1]`
insert	*Ω(1) … O(lg n)*	*O(1)*	sift up
remove the largest	*O(lg n)*	*O(1)*	sift down
change a priority	*O(lg n)*	*O(1)*	if the position is known
bottom-up construction	*O(n)*	*O(1)*	in place
build by *n* insertions	*O(n lg n)*	*O(1)*	worst case: increasing order
heap sort	*O(n lg n)*	*O(1)*	in place
search for an arbitrary item	*O(n)*	*O(1)*	a heap is not a search structure

Table 13.17: Cost of operations in a binary heap with *n* items.

Table 13.17 collects the operations examined in the notes. The table has a common rationale: in a heap the work is bounded by the length of a root-to-leaf path, and by Theorem 13.2 this length is *⌊lg n⌋*. Looking at the largest requires no walking along a path. Bottom-up construction is linear because most paths are short. Search cannot exploit the heap property: if the sought value is smaller than the root, it may be in any subtree. The extra memory is constant in every row, because the heap is the array itself and the routines use only a few index variables.

13.11 Common mistakes

1. **Mixing up the 1-based and 0-based formulas.** In a 1-based array the parent is `i/2` and the children are `2i` and `2i+1`; in a 0-based array the parent is `(i-1)/2` and the children are `2i+1` and `2i+2`. If they are mixed up, the wrong nodes are compared and the heap silently breaks.
2. **Declaring the array one smaller than the capacity.** With `element heap[MAX_ELEMENTS - 1]`, the 199th item is written to `heap[199]`, outside the array.

3. **Writing `return 0;` in a routine that returns `element`.** An integer does not convert to a structure; it is a compile error. Success is returned as a `bool` and the item through a pointer parameter.
4. **Forgetting the `j < n` test and reading `heap[j+1]`.** The cell read is outside the heap; in `build_heap` it may even be outside the array.
5. **Choosing the first child instead of the larger child in deletion.** When the smaller child moves up, it becomes the parent of its larger sibling and the heap property breaks.
6. **Taking a heap for a sorted array or a binary search tree.** Only `heap[1]` is guaranteed to be the largest; there is no order among siblings, and searching a heap takes *O(n)*.
7. **Not stopping the insertion loop at the root.** If `i >= 1` is written instead of the condition `i != 1`, `heap[0]` is read and the new item may be written outside the heap.
8. **Sifting in the wrong direction when a priority changes.** In a max heap, an item whose priority rises is sifted up and one whose priority falls is sifted down. Calling `increase_key` with a smaller value may break the heap.

13.12 Exam-style question

Example 13.4

Into a max heap with `heap[1..6] = 14, 12, 7, 10, 8, 6` initially, 15, 109, 107, 3, 15 are inserted in turn; then `delete_max_heap` is called twice. (i) What are the contents of the array after the insertions? (ii) What are the removed items and the final array? (iii) Which insertion made the fewest shifts and which the most?

(i) The five insertions were traced step by step in Table 13.4. The final array is *109, 107, 14, 15, 15, 6, 7, 10, 12, 3, 8*, *n = 11*. In an exam it is enough to write the path of each insertion: the new position *n + 1*, then comparison with the parent and, if needed, a climb to the parent.

(ii) The first removal returns the 109 at the root. The last item is the 8 at position 11; *n* drops to 10. The children of the root are 107 and 14; 107 moves up to the root and the hole moves down to 2. The children of 2 are the two 15s at positions 4 and 5. Since the left child is not smaller than the right one, `j` stays at 4; since *8 < 15*, the 15 at position 4 moves up and the hole moves down to 4. The children of 4 are 10 and 12; 12 is larger and *8 < 12*, so 12 moves up and the hole moves down to 9. The left child of 9 would be at position 18; *18 > 10*. 8 is written at position 9. The array becomes *107, 15, 14, 12, 15, 6, 7, 10, 8, 3*.

The second removal returns 107. The last item is the 3 at position 10; *n* drops to 9. The children of the root are 15 and 14; 15 moves up to the root and the hole moves down to 2. The children of 2 are 12 and 15; the right one is larger, 15 moves up, and the hole moves down to 5. The left child of 5

would be at position 10, and *10 > 9*; 3 is written at position 5. The removed items are 109 and 107; the final array is *15, 15, 14, 12, 3, 6, 7, 10, 8*, *n = 9*.

(iii) The insertion of 3 made no shift: its parent was not smaller than 8. This is the best case, *Ω(1)*. 109 climbed all the way to the root and made 3 shifts; in a heap of 8 items this is the worst case, equal to the height, *O(lg n)*. All arrays in this solution are identical to the output of a test program that uses the code in the slides verbatim.

13.13 Exercises and solutions

The six exercises of the slides are given here with their solutions. The arrays in the solutions were verified with test programs.

13.13.1 Five insertions into an empty heap

Question. If 20, 5, 30, 10, 40 are inserted into an empty max heap, what is the array after each insertion?

Solution. 20 is written at the root: *20*. 5 goes to position 2 and its parent is smaller than 20: *20, 5*. 30 goes to position 3; its parent is smaller than 30, so 20 moves down and 30 climbs to the root: *30, 5, 20*. 10 goes to position 4; its parent, the 5 at position 2, is smaller, so 5 moves down; its new parent is larger than 10 (it is 30), and 10 stops at position 2: *30, 10, 20, 5*. 40 goes to position 5; its parent is smaller than 40, as is the next one, and it climbs to the root: *40, 30, 20, 5, 10*.

13.13.2 Deletion for a min heap

Question. If `delete_max_heap` is rewritten for a min heap, which lines change?

Solution. The name of the routine becomes `delete_min_heap`. In choosing the child, the smaller child is sought: `if (j < n && heap[j].x > heap[j + 1].x) j++;`. The stopping condition becomes `if (temp.x <= heap[j].x) break;`. Everything else is the same. The full code is given in 13.5.2.

13.13.3 Insertion into a heap of a thousand items

Question. In a heap with *n = 1000* items, how many shifts can an insertion make at most?

Solution. The number of levels of a complete binary tree with 1000 items is *⌈lg 1001⌉ = 10*: $2^9 = 512 \le 1000 < 1024$. Between a leaf and the root there are at most 9 edges. An item that climbs from the leaf level to the root makes one shift per edge; at most 9 shifts, that is, *⌊lg 1000⌋ = 9*. If the new item goes to position 1001, the path is 1001 → 500 → 250 → 125 → 62 → 31 → 15 → 7 → 3 → 1, and it really is 9 edges.

13.13.4 The relatives of node 13

Question. In a 1-based heap, at which indices are the parent and children of node 13?

Solution. The parent is $\lfloor 13/2 \rfloor = 6$. The children are $2 \cdot 13 = 26$ and 27; they exist only if the heap has enough items ($n \geq 26$ for the left child, $n \geq 27$ for the right child). The detailed computation is in Example 13.1.

13.13.5 Construction versus repeated insertion

Question. If repeated `insert` is applied to the array *5, 3, 8, 9, 1, 12, 7* instead of `build_heap`, is the result the same?

Solution. No. `build_heap` gives *12, 9, 8, 3, 1, 5, 7*. Repeated insertion proceeds as follows: *5*; *5, 3*; *8, 3, 5*; *9, 8, 5, 3*; *9, 8, 5, 3, 1*; *12, 8, 9, 3, 1, 5*; *12, 8, 9, 3, 1, 5, 7*. Both results are valid max heaps, but they differ. The same set of values has more than one heap arrangement.

13.13.6 Where is the smallest item?

Question. Where can the smallest item of a max heap be? What does it cost to find it?

Solution. If the values are distinct, the smallest item can be only at a leaf. A node with a child is larger than at least one value, its child, by the heap property; it cannot be the smallest. The leaves are the indices from $\lfloor n/2 \rfloor + 1$ to n. If there are duplicate values, the smallest value may also be found at an internal node, but a copy of that value is certainly at a leaf as well. In both cases it suffices to scan the leaves. There are about $n/2$ leaves and all of them must be examined, because there is no order among the leaves. The cost is $\Theta(n)$. A heap makes finding the largest $O(1)$, but does not speed up finding the smallest.

13.14 Summary

- A **priority queue** is a queue in which the first item to remove is determined by priority rather than arrival order: in the max version the item with the highest value is removed first, in the min version the item with the lowest value.
- As an abstract data type it has insert, remove and peek operations; it can be implemented with an unsorted or sorted array, a list, or a heap.
- A **heap** is the combination of a complete binary tree and the rule “parent ≥ child”. In a 1-based array the parent is `i/2` and the children are `2i` and `2i+1`; no pointers are needed.
- A heap is not a binary search tree; there is no order among siblings. Only the root is guaranteed to be the largest.
- **Insertion**: the first empty position is opened, larger ancestors are shifted down one level each, and the item is written once at the end. Best case $\Omega(1)$, worst case $O(\lg n)$.

- **Deletion**: the root is taken, a place is sought for the last item starting from the root, and larger children are moved up: *O(lg n)*. The test `j < n` checks that the right child exists.
- The height of a complete binary tree is $\lfloor \lg n \rfloor$; this is why heap operations are logarithmic.
- Using a structure instead of a global array allows multiple heaps and a capacity chosen at run time. For a min heap the comparisons are reversed. An item whose priority changes is sifted up or down in *O(lg n)* if its position is known.
- Simple structures push one of the operations to $\Theta(n)$; a heap keeps both logarithmic. In the measurement, at a hundred thousand items the heap came out about 700 times faster than the unsorted array.
- **Bottom-up construction** turns an array into a heap in place and in *O(n)* time; repeated insertion takes *O(n lg n)* in the worst case.
- The **d-ary heap** makes the tree shallower and speeds up insertion; binomial and Fibonacci heaps speed up merging and decrease-key. Heap sort is the application of the priority queue to sorting: *O(n lg n)*, in place.

13.15 Key terms

priority queue abstract data type (ADT) key complete binary tree parent level height heap max-heap min-heap binary search tree up-heap (sifting up) down-heap (sifting down) bottom-up heap construction (heapify) decrease key Fibonacci heap heap sort in-place sort amortized analysis cache heap memory

---

13.16 Test Yourself

1. The values 4, 9, 2, 7, 11 are inserted in turn into an empty max heap. What is the final array? How many shifts are made?

   Answer

   Arrays: *4*; *9, 4*; *9, 4, 2*; *9, 7, 2, 4*; *11, 9, 2, 4, 7*. Shifts: 1 for 9 (4 moves down), 0 for 2, 1 for 7 (4 moves down), 2 for 11 (7 and 9 move down); 4 in total.

2. `delete_max_heap` is called once on the heap *50, 30, 40, 10, 20, 35*. What are the removed item and the new array?

   Answer

   50 is removed. The last item, 35, is taken out, $n = 5$. The children of the root are 30 and 40; 40 moves up to the root and the hole moves down to 3. The left child of 3 would be at position 6, but $6 > 5$. 35 is written at position 3: *40, 30, 35, 10, 20*.

3. In a 1-based heap of 100 items, what are the parent, the children and the level of node 37?

Answer

The parent is $\lfloor 37/2 \rfloor = 18$ and the children are 74 and 75 (both ≤ 100, so they exist). The level is $\lfloor \lg 37 \rfloor = 5$, because $32 \leq 37 < 64$.

4. What does `build_heap` turn the array *1, 2, 3, 4, 5, 6, 7* into? How many moves are made?

Answer

$k = 3$: the children of 3 are 6 and 7; 7 moves up, 3 moves down. $k = 2$: the children of 2 are 4 and 5; 5 moves up, 2 moves down. $k = 1$: the children of the root are 5 and 7; 7 moves up; then, between 6 and 3, 6 moves up; 1 sinks to position 6. The result is *7, 5, 6, 4, 2, 1, 3*; 4 moves.

5. The value 2 is inserted into the min heap *3, 6, 7, 14, 8, 12, 15*. What are the path and the final array?

Answer

The new position is 8. The ancestors are 14 (position 4), 6 (position 2) and 3 (the root); since 2 is smaller than all of them, all three move down. The path is 8 → 4 → 2 → 1, 3 shifts. The final array is *2, 3, 7, 6, 8, 12, 15, 14*.

6. At which indices can the second and third largest items be in a max heap with distinct values?

Answer

The parent of the second largest item can only be the root, because its parent must be larger than it and the only item larger than it is the root: index 2 or 3. The parent of the third largest item is the root or the second largest: index 2, 3, or one of the children of the second largest, that is, one of 4, 5, 6, 7.

7. A new item is inserted into a max heap of 31 items. What is the most shifts that can be made? The fewest?

Answer

The new item goes to position 32; this is the first position of a new level. The path is 32 → 16 → 8 → 4 → 2 → 1, so at most $\lfloor \lg 32 \rfloor = 5$ shifts. If the new item is smaller than or equal to its parent at position 16, no shift is made: at least 0.

8. In a 0-based 3-ary heap, at which indices are the parent and children of node 5?

   Answer

   The parent is $\lfloor (5-1)/3 \rfloor = 1$. The children are $3 \cdot 5 + 1 = 16$, 17 and 18.

9. A million items will first be inserted into a priority queue, and then only the ten largest will be removed. Is an unsorted array or a heap more suitable?

   Answer

   In an unsorted array a million insertions are cheap (each $\Theta(1)$), but each of the ten removals needs about a million comparisons: about ten million in total. For a heap, the items can first be put into an array and turned into a heap with `build_heap` in fewer than n moves; ten removals need about $10 \cdot 2 \cdot 19 = 380$ comparisons. The unsorted array makes about 11 million steps in total and the heap about 2 million (about $1.9n$ comparisons for construction). The heap is still ahead, but the difference is only about a factor of five, because most of the work is done at the input. As the number of removals grows, the heap's advantage grows.

13.17 Further reading

- E. Horowitz, S. Sahni and S. Anderson-Freed, *Fundamentals of Data Structures in C*: the source followed in these notes for heaps, the `adjust` routine and heap sort.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms*, Chapter 6: gives the $O(n)$ analysis of heap construction, heap sort and the priority queue operations with proofs; the chapter on Fibonacci heaps is the source of the amortized times.
- R. Sedgewick, *Algorithms in C*, Parts 1–4, Chapter 9: describes array, list and heap implementations of the priority queue and the indirect heap that tracks items by index.
- J. W. J. Williams, "Algorithm 232: Heapsort", *Communications of the ACM* 7(6), 1964 and R. W. Floyd, "Algorithm 245: Treesort 3", *Communications of the ACM* 7(12), 1964: two short original papers that introduced the binary heap and heap sort and added bottom-up construction.

13.18 References

1. E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, Section 5.6 (Heaps), Chapter 7 (Heap Sort).
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms* (CLRS), MIT Press, Chapter 6 (Heapsort, priority queues) and the chapter on Fibonacci heaps.
3. R. Sedgewick, *Algorithms in C*, Parts 1–4, Chapter 9 (Priority Queues and Heapsort).
4. J. W. J. Williams, "Algorithm 232: Heapsort", *Communications of the ACM*, 7(6), 347–348, 1964.

5. R. W. Floyd, “Algorithm 245: Treesort 3”, *Communications of the ACM*, 7(12), 701, 1964.

PART IV

Hashing, Graphs and Sets

Chapters 14–18

PART IV · CHAPTER 14

Hash Tables

Binary search in a sorted array and search in a balanced search tree proceed by comparing keys with one another. For this reason, in a structure of n items they cannot get below the order of $lg\ n$ steps. A hash table takes a different route: it does not compare keys, it computes the position of the item in the array from the key. Because the computation takes only a few arithmetic operations, search, insertion and deletion finish in constant time on average. These notes first build the simplest form of this idea, direct addressing. They then work out, one by one, the hash functions that reduce a key to a small range of indices: division, multiplication, folding, mid-square, and the polynomial hash for strings. Two different keys may land in the same position; there are two main ways to resolve such a collision. In open addressing the item goes to another empty slot of the table; linear probing, quadratic probing and double hashing search for this slot in different orders. In chaining, every bucket is the head of a linked list. Each method is written in C and its code is traced step by step. The last sections treat the pitfall of deletion in open addressing, the effect of the load factor on cost, and growing the table.

LEARNING OUTCOMES

Explaining the idea
Define a hash table, which searches in *O(1)* on average by computing the position from the key; state how it differs from direct addressing and what it costs.

Choosing a function
Compute division, multiplication, folding, mid-square and string hashes by hand; list the properties of a good hash function; show why the table size is chosen to be prime.

Resolving collisions
Apply linear probing, quadratic probing, double hashing and chaining step by step on a small table; distinguish primary from secondary clustering.

Writing it in C
Write the insertion, search and deletion routines with `malloc` / `free`, `NULL` checks and the full-table case; leave no memory leak.

Deleting correctly
Show why a deleted slot cannot simply be left empty in open addressing and how a deleted marker protects search.

Analyzing
Explain the effect of the load factor on the expected number of probes with formulas and measurements; compute the amortized cost of doubling the table.

14.1 The hashing idea

Consider a student records program. Given a student number, the program must find that student's record. The records are queried thousands of times a day; new students are added and graduates are deleted. Which data structure should be chosen for this job? The section begins with the costs of the familiar structures and shows why a hash table is different.

14.1.1 Reducing the cost of search to a constant

If the records are kept in an unsorted array, insertion is very cheap: the new record is written at the end of the array. Search, however, is expensive. The number sought may be anywhere in the array, or not there at all. In the worst case, therefore, all n records are examined. Deletion also begins with a search, so it is $O(n)$. If the array is kept sorted by number, search drops to $O(\lg n)$ with binary search. This time insertion and deletion become expensive: so that the order is not broken, all records after the new record's position are shifted by one slot. A balanced search tree does all three operations in $O(\lg n)$ and also allows traversing the records in sorted order. Table 14.1 compares the four structures.

Structure	Search	Insert	Delete	Comment
Unsorted array	$O(n)$	$O(1)$	$O(n)$	append at the end, scan from the start
Sorted array	$O(\lg n)$	$O(n)$	$O(n)$	binary search and shifting
Balanced search tree	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$	also gives sorted traversal
Hash table	$O(1)$*	$O(1)$*	$O(1)$*	* average; worst case $O(n)$

Table 14.1: Cost of search, insertion and deletion in four structures. The asterisk on the hash table marks the average case; the worst case is $O(n)$.

The first three structures share a limit: all of them search by comparing keys. Each comparison gives two outcomes, “smaller” or “larger”. To choose one among n possible positions, at least $\lg n$ two-outcome decisions are needed. A comparison-based structure therefore cannot get below $O(\lg n)$. A hash table gets past this limit by giving up comparison. The position is computed directly from the key: position = h(key). The computation is a few arithmetic operations and does not depend on n.

The gain has a price. A hash table does not keep items in key order; positions are scattered according to the result of the computation. Some queries are therefore not cheap in a hash table: “print all records in order of number”, “find the smallest number”, or “list the records whose numbers lie between 1000 and 2000”. For these queries the whole table must be scanned and, if necessary, sorted separately. The second price is the worst case. If all keys fall into the same position, the table turns into a long list and search becomes *O(n)*. That the average case stays constant depends on a good hash function and on keeping the table sufficiently empty. The rest of the notes examine these two conditions.

14.1.2 Direct addressing

The simplest form of the hashing idea is to use the key itself as the array index. If the keys are integers between 0 and 9, an array of ten cells is enough: the record with key k is written in cell `T[k]`. To search, it is enough to look at `T[k]`; there is no loop and no comparison.

Definition 14.1

The set of all values that keys can take is called the **key universe** and is denoted U. When $U = \{0, 1, \ldots, |U| - 1\}$, building an array `T` of $|U|$ cells and keeping the item with key k in cell `T[k]` is called **direct addressing**. Each cell of the array either holds an item or is empty.

Figure 14.1 shows this arrangement. The universe consists of ten keys, and the keys in use are only 2, 3, 5 and 8. Each record is in the cell of its own number; the other six cells are empty.

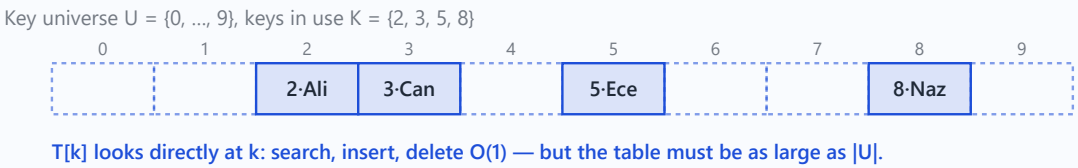


Figure 14.1: A direct-address table. The universe is $U = \{0, \ldots, 9\}$ and the keys in use are $K = \{2, 3, 5, 8\}$. `T[k]` holds the record with key k directly; cells with dashed outlines are empty.

The C code in the slides represents each cell with a small structure. The field `used` says whether the cell is occupied, and the field `name` holds the record's data.

```c
typedef struct { int used; char name[16]; } rec;
rec T[10];              /* key = index */

rec *da_search(int k) {
  return T[k].used ? &T[k] : NULL;
}
```

The code is short, but every line carries a decision. The first line defines a structure type named `rec`. The structure has two fields: `used` of type `int`, and the 16-character array `name`. Thanks to `typedef`, the type name is used without writing `struct` every time. The second line defines the ten-cell array `T` as a global. In C, global variables are filled with zeros when the program starts. So initially the `used` field of every cell is 0, that is, the table is empty. No separate initialization loop needs to be written.

The function `da_search` takes the key `k` and looks at the field `T[k].used`. If the field is nonzero, the cell is occupied and the function returns the address of that cell, `&T[k]`. If the field is zero, there is no record and `NULL` is returned. The conditional expression (`? :`) writes these two cases on one line. That the function returns the address of the record rather than a copy is a deliberate choice. The caller can read or modify the record through this address. The not-found case is also reported explicitly with `NULL`. Insertion and deletion are one-liners in the same way: insertion sets `T[k].used = 1` and writes the data; deletion sets `T[k].used = 0`. All three operations are *O(1)*.

In the test program, cells 2, 3, 5 and 8 were filled with the names Ali, Can, Ece and Naz. `da_search` returned the corresponding names for these four keys and `NULL` for the other six. On the same machine `sizeof(rec)` came out as 20 bytes (a 4-byte `int` and a 16-byte array), and the ten-cell array occupied 200 bytes.

An assumption is hidden in the code: `k` is always between 0 and 9. A `k` outside this range reads memory beyond the bounds of the array. C does not catch this error; the result is undefined behavior. In a real program a check such as `if (k < 0 || k >= 10) return NULL;` should be added at the start of the function.

The problem with direct addressing is the size of the universe. If student numbers have nine digits, $|U| = 10^9$. With 20-byte records this table takes 20 gigabytes. A university may have perhaps 20,000 students: only two hundred-thousandths of the table would be occupied, and the rest would sit empty. When the key is a string, for example a variable name or an e-mail address, the universe is practically infinite. In that case direct addressing cannot be built at all.

The solution is a function that reduces the key to a small range. The size of the table is chosen according to the number of keys in use, for example m = 40,000 cells. Every key is converted by this function to an index between 0 and $m - 1$. This function is called a hash function.

14.1.3 The hash function and basic terms

Definition 14.2

A **hash function** is a function $h: U \to \{0, 1, \ldots, m - 1\}$ from the key universe to the indices of the table. A structure consisting of an array of m cells and a hash function, which looks for the item with key k in the cell with index $h(k)$, is called a **hash table**. Each cell of the table is called a **bucket** or a **slot**. The value $h(k)$ is the **home address** of the key k.

The word “hash” in English means to chop up or mince. The function cuts the key into pieces, mixes them, and produces a small number. This course uses the word “hash” throughout.

A hash function maps $|U|$ values to m indices, and m is much smaller than $|U|$. So some different keys are bound to land on the same index. This is called the pigeonhole principle: if eleven pigeons go into ten holes, at least one hole contains two pigeons. Figure 14.2 shows a function mapping four name keys to six buckets. Ali and Ece have the same home address, 3.

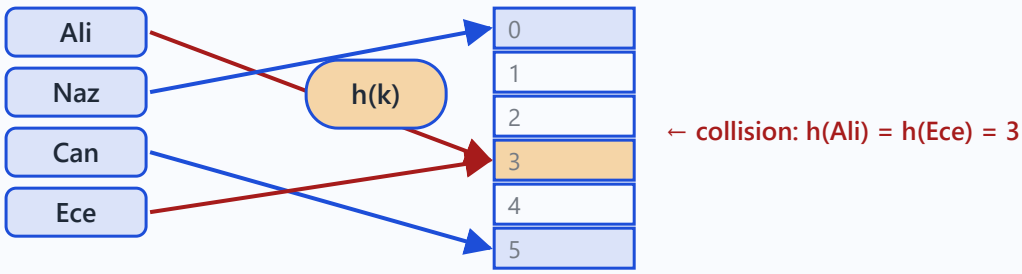


Figure 14.2: Mapping keys to buckets with a hash function. Naz goes to 0 and Can to 5; Ali and Ece both land on 3. This situation is called a collision.

Definition 14.3

When $k_1 \neq k_2$ but $h(k_1) = h(k_2)$, this is called a **collision**. Keys that land on the same home address are called **synonyms**. If the table has n items and m buckets, the ratio $\alpha = n / m$ is called the **load factor** of the table.

Table 14.2 gives the terms together. The name “synonym” comes from linguistics: just as two words are spelled differently but carry the same meaning, two keys are different but arrive at the same address. The load factor measures how full the table is. $\alpha = 0.5$ says that half of the table is full. In some collision-resolution schemes a bucket can hold more than one item, and α can exceed 1.

Term	Meaning
bucket / slot	a position of the table, $0 \ldots m - 1$
home address	$h(k)$: the first place where the key is looked for
collision	$k_1 \neq k_2$ but $h(k_1) = h(k_2)$
synonyms	keys that land on the same home address
load factor α	n / m, the fraction of the table that is full

Table 14.2: Basic terms of a hash table.

As long as $m < |U|$, collisions are unavoidable. Hash table design then comes down to two questions. The first is how to choose a good h: keys should be spread over the buckets as evenly as possible, with few collisions. The second is what to do when a collision occurs: where is the second item that lands

on the same address put, and how is it found during search? Hash functions answer the first question; open addressing and chaining answer the second.

14.1.4 The dictionary abstract data type

A hash table is an implementation. The abstract data type it implements is the dictionary. An abstract data type defines the values and operations of a structure independently of how those operations are written. A dictionary resembles an everyday dictionary: a word (the key) is given and its meaning (the value) is found.

> **Definition 14.4**
>
> A **dictionary** is a collection of key–value pairs in which every key occurs at most once. Its basic operations are insertion, search by key, and deletion by key.

```c
typedef struct { int key; /* + value */ } item;

bool hash_insert(item x);   /* update if found */
item *hash_search(int key); /* NULL if absent  */
bool hash_delete(int key); /* false if absent */
```

The interface in the slides gives the three operations as C declarations. The structure `item` holds the key; the “+ value” in the comment says that in a real application other fields would be added to the structure. `hash_insert` takes an item and returns a `bool` reporting whether the operation succeeded. If the key already exists, the new item is not added; the value of the existing one is updated. This preserves the rule “every key at most once”. `hash_search` returns the address of the item; if the key is absent it gives `NULL`. `hash_delete` returns `true` if it finds and deletes the key, and `false` if it cannot find it. `bool`, `true` and `false` are in the header `<stdbool.h>` introduced with C99; the code does not compile without including this header.

Dictionaries appear in many places. A compiler keeps every variable name in the source code in a **symbol table**: the name is the key, and its type and place in memory are the value. Horowitz's textbook opens hash tables with this example. The compiler looks up many names on every line; if lookup finishes in constant time, compilation speeds up. Web browser caches keep the address as the key and the page as the value. To find repeated items in a list, each item is added to a set; before it is added, the set is searched. A set is a dictionary without values.

Python's `dict`, Java's `HashMap` and C++'s `unordered_map` are all hash tables. C's standard library, however, has no ready-made hash table. The POSIX standard provides a limited routine called `hsearch`: only one table can be created at a time, and items cannot be deleted from it. A C programmer must therefore know how to write a hash table. The remaining sections of these notes do exactly that.

14.2 Hash functions

The speed of a hash table depends on how evenly the keys are spread over the buckets. A function that throws all keys into the same bucket is also a valid function, but it turns the table into a single long list. This section first lists what is expected of a good function, and then works out the common methods for integer and string keys.

14.2.1 Properties of a good hash function

Four properties are sought in a good hash function.

- **It must be fast.** The function runs on every search, insertion and deletion. The computation should finish in a few arithmetic operations. If the key is a string, each character should be processed at most once; that is, the cost should be linear in the key length.
- **It must spread keys uniformly.** Keys should fall into the m buckets in numbers as equal as possible. Theoretical analyses accept this as an assumption: each key falls into each of the m buckets with probability $1/m$, independently of the other keys. This assumption is called the **simple uniform hashing** assumption.
- **It must use all the bits of the key.** Real data is usually patterned. Prices end in 5 or 0, memory addresses are multiples of 8, student numbers begin with the same year codes. A function that looks only at the last digit or the last few bits carries these patterns into the buckets and clusters the keys in a few buckets.
- **It must be deterministic.** The same key must go to the same bucket on every call. Otherwise an item that was inserted once could not be found later. Randomness can be used only when the function is chosen: when the program starts, one function is picked at random from a family of functions and used for the rest of the program.

Uniform distribution cannot be guaranteed without knowing the key set. For every fixed hash function, a key set can be found in which all keys fall into the same bucket. The aim of the methods is to behave well on the patterned data seen in real life.

14.2.2 The division method

The simplest and most common method is to divide the key by the table size and take the remainder.

> **Definition 14.5**
>
> The **division method**: $h(k) = k \bmod m$. Because the remainder is always between 0 and $m - 1$, the result is directly an index.

```c
unsigned h_div(unsigned k, unsigned m) { return k % m; }
```

The function is a single `%` operation. It matters that the parameters are `unsigned`; the reason is given at the end of this subsection. All the subtlety of the method lies in the choice of *m*. Table 14.3 distributes eight keys with two different table sizes.

k	10	20	30	40	50	60	17	27
k mod 10	0	0	0	0	0	0	7	7
k mod 11	10	9	8	7	6	5	6	5

Table 14.3: Home addresses of the same keys with *m = 10* and *m = 11*.

When *m = 10*, *k mod 10* is the last decimal digit of *k*. The function never sees the other digits. The keys 10, 20, ..., 60 all end in 0 and all fall into bucket 0; 17 and 27 also collide in bucket 7. Eight keys are spread over only two buckets. When *m = 11*, the same eight keys go to six different buckets. There are still two collisions (50 and 17 in bucket 6, 60 and 27 in bucket 5), but the multiples of 10 are now spread over consecutive buckets. The reason is that 11 has no common divisor with 10: *10j mod 11* takes ten different values for *j = 1, ..., 10*.

The same problem is even more pronounced for powers of 2. If $m = 2^p$, then $k \bmod 2^p$ is the lowest p bits of the binary representation of *k*. Because data in a computer is stored as binary numbers, patterns in the last bits are very common. The addresses of 8-byte structures are always multiples of 8, that is, their last three bits are 000. In a table with *m = 16*, these addresses fall only into buckets 0 and 8. For this reason the following rule is recommended (CLRS): *m* should be chosen as a prime number not close to a power of 2. For example, for about 2000 keys, 701 is a good choice: it is prime and far from 512 and 1024. In this table about three keys fall into each bucket on average.

The division method has one more trap in C. In C the result of the `%` operator carries the same sign as the dividend: the expression `-7 % 11` is −7, not 4. If the key is of type `int` and can be negative, `k % m` gives a negative index. Memory before the start of the array is read or overwritten. The remedy is one of two ways. Either the key is kept `unsigned` from the start (`h_div` in the slides does this) or the remainder is made positive:

```c
unsigned h_safe(int k, int m) {
  int r = k % m;            /* -m < r < m */
  if (r < 0) r += m;        /* same as ((k % m) + m) % m */
  return (unsigned)r;
}
```

In the test program `-7 % 11` really gave −7; `h_safe(-7, 11)` returned 4 and `h_safe(-22, 11)` returned 0. Because the remainder lies between −*m* and *m*, adding *m* once moves it into the range 0 to *m* − *1*. The expression `((k % m) + m) % m` from the slides does the same job without a branch.

Common mistake

Thinking that the mathematical "mod" operation and the C operator `%` are the same. In mathematics $-7 \bmod 11 = 4$; in C `-7 % 11 == -7` . This bug produces no compiler warning; it shows up only when a negative key arrives, and it writes outside the array.

Cost. The division method is a single integer division: $O(1)$. Division is a slower operation on a processor than addition and multiplication; still, since it is done once per key, it does not change the cost of the table.

14.2.3 The multiplication method

The division method needs a good m. The multiplication method reduces this dependence: the mixing is done not by m but by a constant multiplier.

Definition 14.6

The **multiplication method**: for a constant $0 < A < 1$, $h(k) = \lfloor m \cdot (k \cdot A \bmod 1) \rfloor$. Here $x \bmod 1$ is the fractional part of x: $x - \lfloor x \rfloor$.

The computation has two steps. First k is multiplied by A and the integer part of the result is discarded. A fraction between 0 and 1 remains. Then this fraction is multiplied by m and rounded down; the result is an integer between 0 and $m - 1$. The fractional part is affected by all the digits of k. So the choice of m is not critical, and $m = 2^p$ can be chosen freely. Knuth suggests the fractional part of the golden ratio for A: $A \approx (\sqrt{5} - 1)/2 = 0.6180339887\ldots$ This value spreads consecutive keys very evenly over the fractional interval.

Example (CLRS). Take $k = 123456$, $p = 14$, that is, $m = 2^{14} = 16384$. The floating-point computation goes as follows:

> k · A = 123456 · 0.6180339887 = 76300.0041151…
> k · A mod 1 = 0.0041151…
> m · 0.0041151… = 16384 · 0.0041151… = 67.42…
> h(k) = ⌊67.42…⌋ = 67

On a computer, floating-point numbers carry rounding error and are slow. For this reason the method is usually written with integers. Let the word length be $w = 32$ bits and take $s = \lfloor A \cdot 2^{32} \rfloor = 2654435769$. The low 32 bits of the product $k \cdot s$ equal 2^{32} times the fractional part of $k \cdot A$. The top p bits of these 32 bits are the integer part of 2^p times the fractional part. That is exactly the value being sought.

```c
/* w = 32 bit, m = 2^p; s = A * 2^32 */
uint32_t h_mul(uint32_t k, int p) {
  const uint32_t s = 2654435769u;
  return (uint32_t)(k * s) >> (32 - p);
}
```

The code is three lines. `uint32_t` is the exact 32-bit unsigned integer type from the header `<stdint.h>`. The product `k * s` is computed between two `uint32_t` values, and if the result does not fit in 32 bits it overflows. Overflow of unsigned integers is defined in C: the result is the remainder modulo 2^{32}, that is, the upper bits are discarded. The discarded part corresponds to the integer part of the product, and the remaining 32 bits correspond to the fractional part. Finally the shift `>> (32 - p)` discards the low $32 - p$ bits and leaves the top p bits.

On the example: $123456 \cdot 2654435769 = 327{,}706{,}022{,}297{,}664$. The remainder of this number modulo 2^{32} is 17,612,864. Shifting right by 18 bits means dividing by $2^{18} = 262{,}144$ and taking the integer part: $17{,}612{,}864 / 262{,}144 = 67.19\ldots$, so the result is 67. The test program got 67 from both the integer version and the floating-point version. The fractions of the two versions show small differences (67.42 and 67.19), because s is a rounded version of $A \cdot 2^{32}$; their integer parts are the same.

The code has one limit: `p` must be between 1 and 32. If `p = 0` is given, the shift amount becomes 32, and shifting a 32-bit number by 32 bits is undefined behavior in C. Since a table with $m = 1$ is meaningless anyway, this case causes no trouble in practice, but it should be written into the function's contract. **Cost:** one multiplication and one shift, $O(1)$. Because multiplication is faster than division, the method is also advantageous in speed.

14.2.4 Folding and mid-square

If the key is too long to fit in a processor word, for example a 14-digit identity number, it must first be shortened. The folding method does this.

Definition 14.7

Folding: the key is divided from left to right into parts of equal length and the parts are added. If the parts are added as they are, it is called **shift folding**; if the even-numbered parts are reversed before adding, it is called **folding at the boundaries**.

In Horowitz's example the key is $x = 12320324111220$. The three-digit parts are $P_1 = 123$, $P_2 = 203$, $P_3 = 241$, $P_4 = 112$ and $P_5 = 20$; the last part has two digits. Figure 14.3 shows the two sums.

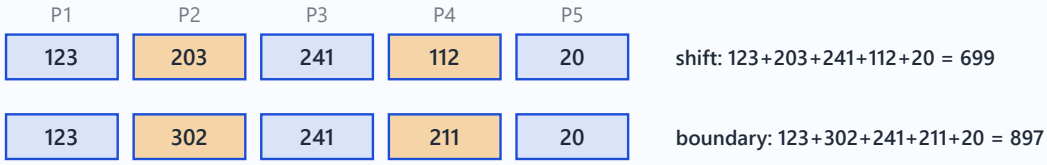


Figure 14.3: Folding the key 12320324111220. Top row, shift folding: the parts are added as they are, 699. Bottom row, folding at the boundaries: the second and fourth parts are reversed (203 → 302, 112 → 211), 897.

In shift folding the sum is *123 + 203 + 241 + 112 + 20 = 699*. In folding at the boundaries, the key is folded back and forth at the part boundaries like a paper strip. When folded, every second part lands on top face down, and its digits are read in reverse. The sum is *123 + 302 + 241 + 211 + 20 = 897*. If the result is larger than the table size, *mod m* is also taken; for example, for *m = 11* both values give 6. The following function computes both kinds with a flag. The key is given as a string of digits, because a 14-digit number does not fit in a 32-bit integer.

```
/* x: digit string; parts of r digits, left to right */
unsigned fold(const char *x, int r, bool boundary) {
  unsigned sum = 0;
  int n = (int)strlen(x);
  for (int start = 0, idx = 0; start < n; start += r, idx++) {
    int end = start + r < n ? start + r : n;   /* part: [start, end) */
    unsigned part = 0;
    if (boundary && idx % 2 == 1)          /* even-numbered part: reversed */
      for (int j = end - 1; j >= start; j--) part = part * 10 + (unsigned)(x[j] - '0');
    else
      for (int j = start; j < end; j++) part = part * 10 + (unsigned)(x[j] - '0');
    sum += part;
  }
  return sum;
}
```

The outer loop increases `start` by `r` each round and moves through the parts in turn; `idx` is the 0-based index of the part. `end` is one past the end of the part; since the last part may be short, the smaller of `start + r` and `n` is taken. The inner loops convert the digits of the part into a number: for each digit, `part = part * 10 + digit`. The expression `x[j] - '0'` converts a digit character to its numeric value; `'7' - '0'` is 7. In folding at the boundaries, the parts with odd `idx` (the second, fourth, ...) are read from right to left. In the test program the function gave the values 699 and 897. Since every digit of the key is processed once, the cost is linear in the key length.

The **mid-square** method squares the key and uses the middle *r* bits of the result as the index; the table size is chosen as $m = 2^r$. Why the middle? The lowest bits of a square depend only on the lowest bits of the number; the highest bits depend largely on the magnitude of the number. The middle bits, on the other hand, are formed in the product with contributions from nearly all bits of *k*. The idea is

easy to see with a decimal example: for $k = 1234$, $k^2 = 1522756$. The middle three of the seven digits are 227; in a table with $m = 1000$ the home address is 227. If any digit of 1234 changes, these three digits change too.

14.2.5 String keys: the polynomial hash

In practice most keys are strings: variable names, user names, words. The first way that comes to mind to convert a string to an integer is to add up the character codes.

```c
unsigned sum_hash(const char *s, unsigned m) {  /* bad example */
  unsigned h = 0;
  for (; *s != '\0'; s++)
    h += (unsigned char)*s;
  return h % m;
}
```

This function is bad. Addition forgets order: “abc” and “cba” consist of the same letters, their sums are $97 + 98 + 99 = 294$, and both fall into the same bucket. Strings obtained by rearranging the letters of a string are called anagrams; the additive hash collides all anagrams. The second problem is the narrowness of the range. The sum of lowercase strings of 8 letters is at most $8 \cdot 122 = 976$. Even if the table has 10,000 buckets, the keys are squeezed into the first thousand buckets.

The solution is to take the position of each character into account as well. A string $c_0\, c_1 \ldots c_{n-1}$ of n characters is read like a number in base 31:

$$h = c_0 \cdot 31^{n-1} + c_1 \cdot 31^{n-2} + \ldots + c_{n-2} \cdot 31 + c_{n-1}$$

This expression is a polynomial. If computed directly, a separate power would be taken for each term. Horner's rule nests the polynomial: $h = (\ldots((c_0 \cdot 31 + c_1) \cdot 31 + c_2) \cdot 31 + \ldots) + c_{n-1}$. One multiplication and one addition per character then suffice. The two functions in the slides use this rule.

```c
unsigned str_hash(const char *s, unsigned m) {
  unsigned h = 0;
  for (; *s != '\0'; s++)
    h = (h * 31 + (unsigned char)*s) % m;
  return h;
}
```

`str_hash` starts `h` at 0. The loop advances the pointer `s` over the characters of the string and stops at the `'\0'` terminator. At each step `h` is multiplied by 31, the code of the new character is added, and the result is taken *mod m*. The cast `(unsigned char)*s` is important: `char` is signed on many compilers, and non-ASCII bytes such as “ç” and “ğ” carry negative values. The cast reads the character as a number between 0 and 255.

Taking the mod at every step has two benefits. First, `h` always stays smaller than m and the product does not overflow (provided that $31 \cdot m + 255$ fits in an `unsigned`). Second, the result does not change: in modular arithmetic $(a \cdot 31 + c) \bmod m$ equals $((a \bmod m) \cdot 31 + c) \bmod m$. Taking the mod at intermediate steps gives the same result as taking it once at the end.

```c
unsigned long djb2(const char *s) {  /* Bernstein */
  unsigned long h = 5381;
  int c;
  while ((c = (unsigned char)*s++) != 0)
    h = h * 33 + c;
  return h;       /* then: h % m */
}
```

The second function is Dan Bernstein's widely used `djb2` hash. It starts with the initial value 5381 and computes $h \cdot 33 + c$ for each character. It takes no mod at intermediate steps; when the `unsigned long` overflows, the result automatically becomes the remainder modulo 2^w (w is the number of bits of the type). Since unsigned overflow is defined in C, this is safe. For a table index, the caller takes the *mod m* of the result. The expression `(c = (unsigned char)*s++) != 0` in the loop condition does three things: it reads the character, advances the pointer, and checks whether the character is zero. On Windows `unsigned long` is 32 bits and on Linux it is 64 bits; so `djb2` may give different numbers for the same string on the two systems. On the test machine (Windows), `djb2("kedi")` came out as 2090432258.

Table 14.4 compares three functions on anagrams. The polynomial hash separates “abc” from “cba”, because “a” is multiplied by 31^2 in one and by 1 in the other.

String	Sum	`sum_hash`	`str_hash`
"abc"	294	8	5
"cba"	294	8	0
"ab"	195	8	3
"ba"	195	8	0

Table 14.4: Additive and polynomial hashes for anagrams ($m = 11$). The values were taken from the test program.

The choice of the number 31 is not random. The base should be odd: an even base fills the lowest bits with zeros during overflow, and the influence of the first characters is lost. The base should also be larger than the number of letters in the alphabet, so that different strings do not give the same “number”. Java's `String.hashCode` method also uses 31. Multiplying by 31 also has a cheap equivalent: $31 \cdot h = 32 \cdot h - h$, that is, a shift and a subtraction.

Cost. All three string functions process each character once. For a key of L characters the time is $O(L)$ and the memory is $O(1)$. The "$O(1)$" cost of a hash table really means "$O(L)$" for string keys. Since the key length is independent of the table size, this is counted as a constant.

14.2.6 Code trace: `str_hash("kedi", 11)`

The character codes of the string "kedi" in the ASCII table are k = 107, e = 101, d = 100 and i = 105. The table has 11 buckets. Table 14.5 shows each round of the loop.

Round	`*s`	Code	Previous `h`	`h * 31 + code`	New `h`
1	`'k'`	107	0	0 · 31 + 107 = 107	107 mod 11 = 8
2	`'e'`	101	8	8 · 31 + 101 = 349	349 mod 11 = 8
3	`'d'`	100	8	8 · 31 + 100 = 348	348 mod 11 = 7
4	`'i'`	105	7	7 · 31 + 105 = 322	322 mod 11 = 3
5	`'\0'`	—	3	the loop condition is false; `return 3`	

Table 14.5: Trace of `str_hash("kedi", 11)`. In each round `h = (h * 31 + *s) % 11`.

In the first round `h` is zero, so the multiplication has no effect, and `h` becomes the remainder of 107 modulo 11: $107 = 9 \cdot 11 + 8$. In the second round the old value is multiplied by 31 and the code of "e" is added: $349 = 31 \cdot 11 + 8$. The third and fourth rounds give 7 and 3. At the fifth check `*s` is the terminator character, the loop ends, and the function returns 3. The home address of the key "kedi" is 3.

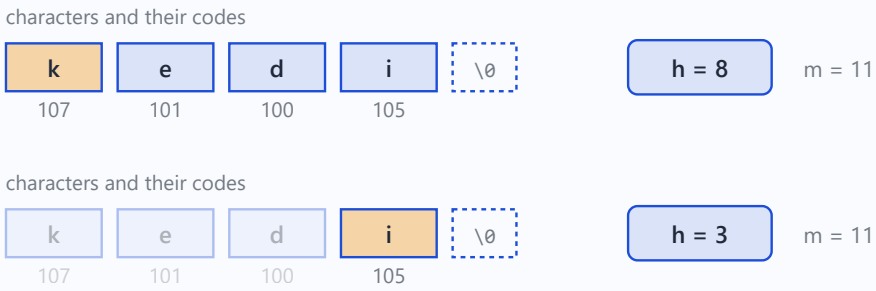


Figure 14.4: Two moments of the trace of `str_hash("kedi", 11)`. Above, the first character has been processed, $h = 8$. Below, the last character has been processed too, $h = 3$; the processed characters are shown faded.

Let us also check the result without taking the mod at intermediate steps. The full polynomial is $107 \cdot 31^3 + 101 \cdot 31^2 + 100 \cdot 31 + 105 = 3{,}187{,}637 + 97{,}061 + 3{,}100 + 105 = 3{,}287{,}903$, and $3{,}287{,}903 \bmod 11 = 3$. The two computations give the same result; that taking the mod at intermediate steps does not change the result can also be seen here with numbers. For a four-character string the loop runs four times and ends at the fifth check: the cost is linear in the string length.

14.3 Open addressing

A good hash function reduces collisions but cannot eliminate them. This section first shows how early collisions begin, and then examines the methods that place the colliding item in another slot inside the table.

14.3.1 Collisions are unavoidable

If the table is large and the number of keys is small, one might expect collisions to be rare. A probability computation refutes this intuition. The famous **birthday paradox** says: in a class of 23 people, the probability that two people share a birthday is about 50.7%. The computation is done through the complement. The probability that all 23 people were born on different days is

> $(365/365) \cdot (364/365) \cdot (363/365) \cdot \ldots \cdot (343/365) \approx 0.4927$

The first person can be born on any day; the probability that the second is born on a different day is 364/365; the third must differ from the first two, and so on. The probability of at least one match is *1 − 0.4927 ≈ 0.507*.

Let us translate this to a hash table: 23 keys, each landing in a random slot, are inserted into a table with $m = 365$ slots. Only 6% of the table is full, but the probability of at least one collision is more than half. When the number of keys reaches 57, this probability exceeds 99%. The general rule is this: in a table of m slots, the first collision is expected after about $\sqrt{m}$ keys. In a table of a million slots this number is around a thousand. Collisions should therefore be treated not as an exceptional case but as the normal case.

There are two basic ways to resolve a collision. Figure 14.5 shows both on the same keys.

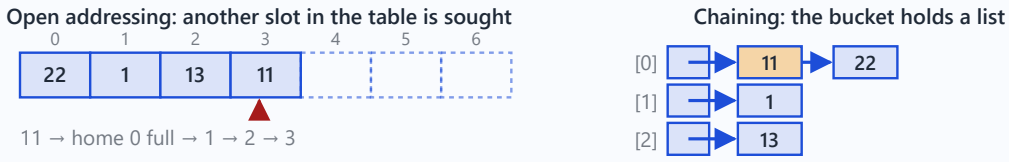


Figure 14.5: Two solutions to collisions. On the left, open addressing: the home address of 11, slot 0, is occupied; since 1 and 2 are occupied too, 11 is written in the first empty slot, 3. On the right, chaining: every bucket is the head of a list; 11 and 22 sit together in the list of bucket 0.

Definition 14.8

Open addressing: all items are kept in the slots of the table, and each slot holds at most one item. If the home address is occupied, other slots are examined according to a certain rule. Checking whether a slot is empty is called a **probe**, and the order of the slots examined is called the **probe sequence**. The probe sequence is written $h(k, i)$: the slot examined at the i-th attempt ($i = 0, 1, \ldots, m - 1$).

In open addressing every item occupies a slot, so $n \le m$, that is, $\alpha \le 1$. When the table is full, no new item can be inserted. In **chaining**, by contrast, every bucket is the head of a linked list; items that fall into the same bucket are added to that list. Since the list can grow as long as needed, $\alpha > 1$ is possible. Chaining is treated in detail in its own section. The rest of this section examines the three probing rules of open addressing: linear probing, quadratic probing and double hashing. All the examples use the same table: $m = 11$, $h(k) = k \bmod 11$, and the keys 22, 1, 13, 11, 24, 33 inserted in this order. The home addresses of these keys are 0, 1, 2, 0, 2, 0: 11 and 33 collide with 22, and 24 collides with 13.

14.3.2 Linear probing

Definition 14.9

Linear probing: $h(k, i) = (h(k) + i) \bmod m$. If the home address is occupied, the next slot is examined; if that is occupied too, the one after it. When the end of the table is reached, probing wraps around to the start.

The rule is like someone looking for a seat in a cinema: if the numbered seat is taken, they sit next to it. The insertion of the six keys proceeds as follows.

1. The home address of 22 is $22 \bmod 11 = 0$. Slot 0 is empty; 22 is written there. One probe.
2. The home address of 1 is 1 and that of 13 is 2. Both land in empty slots; one probe each.
3. The home address of 11 is 0 and it is occupied by 22. Next slot 1 (occupied, 1), then slot 2 (occupied, 13), then slot 3 is examined. Slot 3 is empty. The probe sequence is 0, 1, 2, 3; four probes.
4. The home address of 24 is 2 and it is occupied by 13. Slot 3 now holds 11. Slot 4 is empty. The probe sequence is 2, 3, 4; three probes.
5. The home address of 33 is 0. Slots 0 through 4 are all occupied; slot 5 is empty. The probe sequence is 0, 1, 2, 3, 4, 5; six probes.

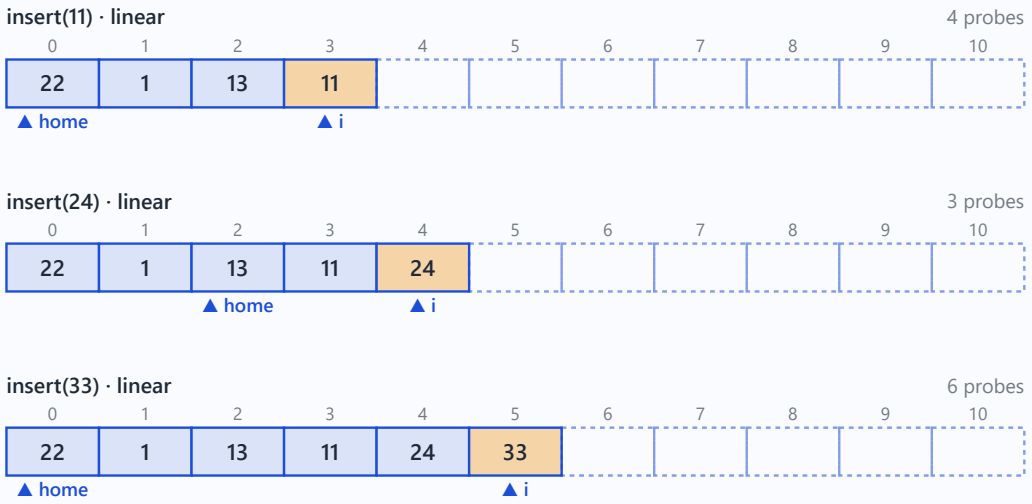


Figure 14.6: Insertion with linear probing. Above, 11 settled in slot 3, three slots away from its home address. In the middle, 24 settled in slot 4, two slots away from its home address, 2. Below, 33 settled in slot 5, five slots away from its home address, 0. Slots 0–5 form an unbroken block of occupied slots.

In the final state the first six slots of the table form a block of occupied slots (Figure 14.6). A new key whose home address falls on any slot inside this block walks to the end of the block and settles there; the block grows by one more slot. As the block grows, the chance of landing in it also grows: a block of 6 slots covers six of the 11 possible home addresses. Big blocks get bigger.

Definition 14.10

In linear probing, the tendency of occupied slots to form long consecutive blocks, and of these blocks to enlarge themselves, is called **primary clustering**.

Primary clustering does not affect only synonyms. In the example, the home address of 24 is 2 and it has nothing to do with 22 or 11. Yet because of 11 it had to walk one more slot. Keys with different home addresses share the same block and slow one another down. As the table approaches full, blocks merge and the average number of probes grows quickly. The numerical value of this growth is given in the load factor section.

14.3.3 C implementation: slot state and insertion

In open addressing it is not enough for a slot to be just “full” or “empty”. For deletion to work correctly, a third state is needed. Why this state is needed is shown in detail in the deletion section; here only its definition is given.

```
#define M 11
typedef enum { EMPTY, USED, DELETED } state_t;
typedef struct {
  int key;          /* assumes key >= 0 */
  state_t state;
} slot;
slot table[M];          /* global: all 0 = EMPTY */
```

`#define M 11` defines the table size as a macro constant. The preprocessor replaces every `M` in the code with 11 before compilation. `enum` defines three constants: `EMPTY` = 0, `USED` = 1 and `DELETED` = 2. If values are not given, C assigns 0 to the first constant and increments for each one after. `state_t` is the type name for these numbers; writing `EMPTY` instead of `0` in the code improves readability.

The structure `slot` has two fields: the key and the state. The assumption in the comment (key ≥ 0) removes the negative-remainder problem of the division method for this code. The array `table` is global and is filled with zeros when the program starts. Because `EMPTY` equals zero, all slots are empty at the start. The order of the `enum` constants therefore matters: if `EMPTY` were not written first, the table would not count as empty at the start.

In memory `table` is a single block of 11 structures laid out one after another. On the test machine each slot is 8 bytes (a 4-byte `int` and a 4-byte `enum`), and the table occupies 88 bytes. There are no pointers and no `malloc`. This is an important advantage of open addressing and is taken up again in the design comparison.

```
int insert(int k) {
  int i = k % M;                  /* home addr */
  for (int d = 0; d < M; d++, i = (i + 1) % M) {
    if (table[i].state != USED) { /* free slot */
      table[i].key = k;
      table[i].state = USED;
      return i;
    }
  }
  return -1;                      /* table full */
}
```

The function `insert` works line by line as follows.

- **Line 2:** `i` is the home address of the key. The first probe will be made here.
- **Line 3:** The `for` loop manages two variables. `d` is the attempt counter and goes from 0 to *M − 1*. `i` is the slot being examined; at the end of every round it is incremented and taken *mod M*, so after 10 it returns to 0. The comma operator performs the two updates in one expression.

- **Lines 4–8:** If the slot is not `USED`, that is, it is `EMPTY` or `DELETED`, the key is written there, the state is set to `USED`, and the index of the slot is returned.
- **Line 10:** If the loop found occupied slots in all *M* attempts, the table is full; the function returns −1.

Limiting the number of attempts to *M* is important. The sequence of linear probing visits every slot once in *M* steps. Without the limit, the loop would run forever on a full table. In the test program, 5 was to be inserted into a table filled with the keys 0, 11, 22, …, 110; `insert` returned −1 after 11 attempts.

The function has a known shortcoming: it does not check whether the key is already in the table. If the same key is inserted a second time, a second copy is written into the first empty slot of the probe sequence. The dictionary rule "every key once" is the caller's responsibility in this code. Once `DELETED` slots come into play the problem becomes subtler; its solution is the function `insert_unique` in the deletion section.

Cost. Insertion takes as long as the number of probes. In the best case the home address is empty: $\Theta(1)$. In the worst case the whole table is traversed: $O(m)$. The average case depends on the load factor; as long as α is kept constant it is $O(1)$. No extra memory is used.

14.3.4 Code trace: `insert(33)`

The table holds 22, 1, 13, 11 and 24; slots 0–4 are a block of occupied slots. Table 14.6 traces the call `insert(33)`.

`d`	`i`	`table[i]`	**Decision**
—	0	—	home address *33 mod 11 = 0*
0	0	22, `USED`	next: *i = (0 + 1) mod 11 = 1*
1	1	1, `USED`	next: *i = 2*
2	2	13, `USED`	next: *i = 3*
3	3	11, `USED`	next: *i = 4*
4	4	24, `USED`	next: *i = 5*
5	5	`EMPTY`	`table[5] = {33, USED}` ; `return 5`

Table 14.6: Trace of `insert(33)`, linear probing, *M = 11*.

The loop runs six rounds; the work in each round is the same: look at the state field and advance `i`. An empty slot is found at the sixth probe. 33 has landed five slots away from its home address. The cause is not 33's own synonyms but the whole block. Of the five keys in slots 0–4, only two (22 and 11) have the same home address as 33.

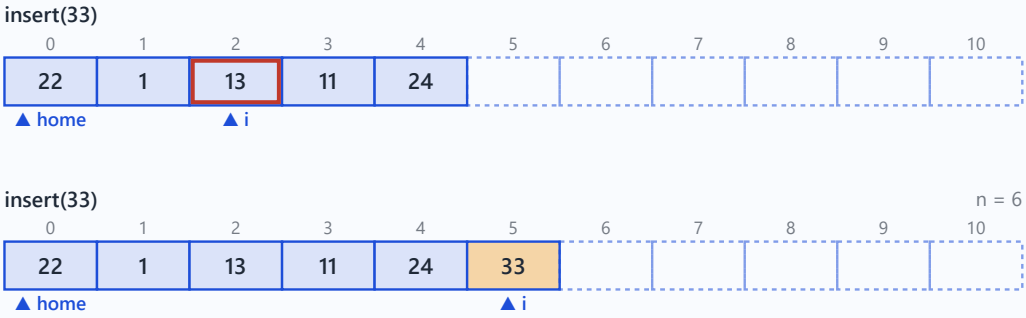


Figure 14.7: Two moments of the trace of `insert(33)`. Above, the 2nd attempt: slot 2 is occupied and `i` advances; the "home" marker shows the home address and "i" the slot being examined. Below, 33 has been written to slot 5.

14.3.5 Search and a code trace

Search follows exactly the probe sequence that insertion followed. If the key is in the table, it is somewhere in this sequence. But how does the search know when to say "absent"? Insertion writes to the first empty slot it meets in the probe sequence. So if the sought key were in the table, it would be found before the first `EMPTY` slot in the sequence. A search that meets an `EMPTY` slot can stop.

```
int search(int k) {
  int i = k % M;
  for (int d = 0; d < M; d++, i = (i + 1) % M) {
    if (table[i].state == EMPTY) return -1;
    if (table[i].state == USED &&
        table[i].key == k) return i;
  }
  return -1;
}
bool erase(int k) {
  int i = search(k);
  if (i < 0) return false;
  table[i].state = DELETED;   /* tombstone */
  return true;
}
```

There are two functions in this code fragment; here `search` is explained, and `erase` is the subject of the deletion section. `search` uses the same loop skeleton as `insert`. At each slot it asks two questions. If the slot is `EMPTY`, the search ends and returns −1. If the slot is `USED` and its key is `k`, the sought key has been found; the index is returned. In the other cases, that is, if the slot is occupied by another key or is `DELETED`, the search moves to the next slot. The condition `table[i].state == USED` in the second `if` is necessary. In a `DELETED` slot the old key still stands in the `key` field; without this condition a deleted key would count as found. If the loop ends after M attempts without meeting any `EMPTY` slot, the table is full and the key is absent; the function again returns −1.

Table 14.7 traces two searches. The table holds 22, 1, 13, 11, 24 and 33 (slots 0–5).

Call	Slots examined	Result	Probes
`search(24)`	2 (13) → 3 (11) → 4 (24)	found, `return 4`	3
`search(44)`	0 (22) → 1 (1) → 2 (13) → 3 (11) → 4 (24) → 5 (33) → 6 (`EMPTY`)	absent, `return -1`	7

Table 14.7: Trace of `search(24)` and `search(44)`. Table: slots 0–5 = 22, 1, 13, 11, 24, 33; slots 6–10 empty.

`search(24)` starts at home address 2. Slot 2 holds 13; the state is `USED` but the key is different, so it continues. Slot 3 holds 11, continue again. In slot 4 it finds 24 and returns 4. Three probes, the same as the three probes made when 24 was inserted. This is no coincidence: a successful search retraces the path the key took when it was inserted.

`search(44)` is longer. The home address of 44 is 0, and the whole block 0–5 is occupied by other keys. Slot 6 is `EMPTY` : if 44 were in the table, it would have been written at slot 6 at the latest, so it is absent. The unsuccessful search took seven probes. Unsuccessful searches are generally longer than successful ones, because they always walk to an empty slot. A search for a key that is not in the table and whose home address falls between 6 and 10, in contrast, ends in a single probe. In the test program `search(5)` was such a search and returned −1 at once.

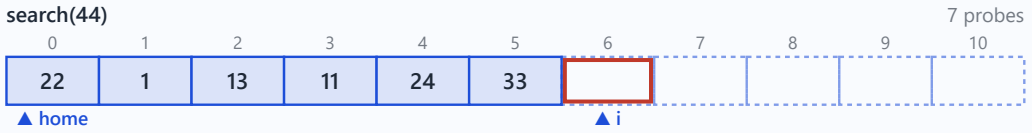


Figure 14.8: The end of the trace of `search(44)` . The search starting at home address 0 walked through the block of occupied slots and met an empty slot at slot 6: 44 is not in the table.

Cost. A successful search takes as many probes as the key took when it was inserted. An unsuccessful search takes as many as the distance from the home address to the first empty slot. Both are $O(m)$ in the worst case and $O(1)$ on average when α is constant.

14.3.6 Quadratic probing

The cause of primary clustering is that the probe sequence passes through adjacent slots. Quadratic probing moves away from the home address in steadily growing steps.

Definition 14.11

Quadratic probing: $h(k, i) = (h(k) + i^2) \bmod m$. The attempts look at slots 0, 1, 4, 9, 16, … away from the home address.

The same six keys settle as follows with quadratic probing. 22, 1 and 13 again land on their home addresses. The probe sequence of 11 is *0, 0 + 1 = 1, 0 + 4 = 4*: slots 0 and 1 are occupied and slot 4 is empty. The sequence of 24 is *2, 2 + 1 = 3*, and since slot 3 is empty it settles in two probes. The sequence of 33 is *0, 1, 4, 9*: slots 0, 1 and 4 are occupied, and 33 is written in slot 9. For 33, linear probing needed six probes; quadratic probing needs only four.

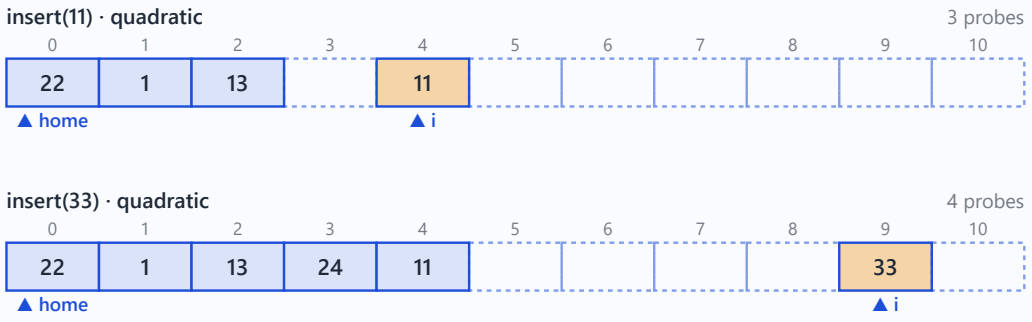


Figure 14.9: Insertion with quadratic probing. Above, 11 settled in slot 4 along the probe sequence 0, 1, 4. Below, 33 settled in slot 9 along the probe sequence 0, 1, 4, 9. The occupied slots no longer form a single block.

The final table is 22, 1, 13, 24, 11 (slots 0–4) and 33 (slot 9). Quadratic probing removes primary clustering: 24 no longer waits behind 11 but settles in the first empty place on its own sequence. But keys with the same home address still follow the same sequence. 22, 11 and 33 all advance along the sequence 0, 1, 4, 9, … and cut each other off. This milder clustering is called **secondary clustering**.

A drawback of quadratic probing is that the probe sequence does not visit all slots. For $m = 11$ and home address 0, the attempts $i = 0, 1, \ldots, 10$ look at slots 0, 1, 4, 9, 5, 3, 3, 5, 9, 4, 1 in turn. The sequence contains only six distinct slots: 0, 1, 3, 4, 5, 9. Slots 2, 6, 7, 8 and 10 are never probed. Insertion can fail even though the table has room. The following result draws the line on this problem.

Theorem 14.1 (An empty slot under quadratic probing)

If m is prime, the first $\lceil m/2 \rceil$ probes of quadratic probing look at distinct slots. Therefore, if the table is at most half full ($\alpha \le 1/2$), an empty slot is always found for a new key.

Proof. Suppose two probes with $0 \le i < j < \lceil m/2 \rceil$ look at the same slot. Then $h + i^2 \equiv h + j^2 \pmod m$, that is, m divides the product $j^2 - i^2 = (j - i)(j + i)$. Since m is prime, it must divide one of the factors. But $0 < j - i < m$ and $0 < j + i < m$ (the latter is the sum of two numbers smaller than $m/2$). So neither factor can be a multiple of m. Contradiction: the first $\lceil m/2 \rceil$ probes look at distinct slots. If the table has fewer than $m/2$ occupied slots, at least one of these distinct slots is empty. ■

For $m = 11$, $\lceil 11/2 \rceil = 6$, and indeed the first six probes, 0, 1, 4, 9, 5, 3, look at distinct slots. The practical consequence of the theorem is simple: a table that uses quadratic probing is given a prime size and is not filled beyond half.

14.3.7 Double hashing

The cause of secondary clustering is that the probe sequence depends only on the home address. Double hashing also computes the step size from the key.

Definition 14.12

Double hashing: with two hash functions h_1 and h_2, $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$. h_1 gives the home address and h_2 gives the step size.

In the example, $h_1(k) = k \bmod 11$ and $h_2(k) = 1 + (k \bmod 10)$ are chosen. The step sizes are 2 for 11, 5 for 24 and 4 for 33. 22, 1 and 13 land on their home addresses. The sequence of 11 is *0, 2, 4*: slots 0 and 2 are occupied and 11 is written in slot 4. The sequence of 24 is *2, 7*; slot 7 is empty. The sequence of 33 is *0, 4, 8*: slot 0 holds 22 and slot 4 holds 11; 33 is written in slot 8. Three probes sufficed for 33.

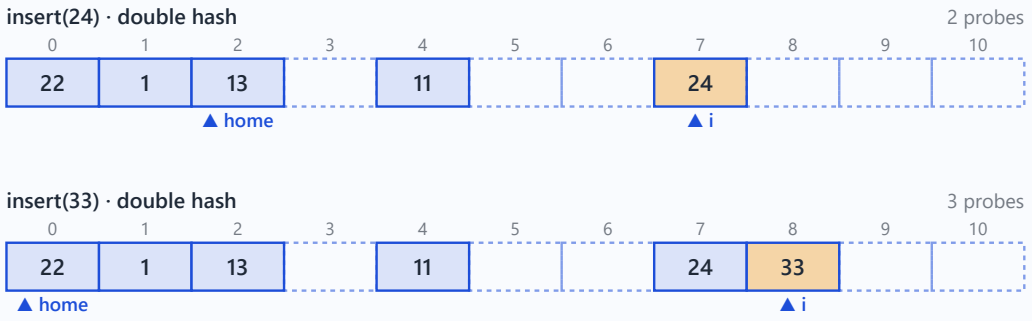


Figure 14.10: Insertion with double hashing. Above, 24 jumped from slot 2 to slot 7 with step size 5. Below, 33 followed the sequence 0, 4, 8 with step size 4 and settled in slot 8.

Here 22, 11 and 33 have the same home address but different step sizes (3, 2 and 4, respectively). The three follow different probe sequences and the chance that they cut each other's path is reduced. In linear and quadratic probing the probe sequence depends only on the home address; m different home addresses give m different sequences. In double hashing the sequence depends on the pair of home address and step size. The number of such pairs is of the order of the square of m.

h_2 must satisfy two conditions. First, it must never be 0. If $h_2(k) = 0$, all probes would look at the home address and the probing would never advance. That is why h_2 begins with "1 +". Second, $h_2(k)$ and m must be relatively prime. For example, if $m = 10$ and the step size were 6, the sequence starting at slot 5 would go around as 5, 1, 7, 3, 9, 5, ...; only five of the ten slots would be probed. If m is chosen prime, every step size between 1 and $m - 1$ is relatively prime to m and the condition holds automatically. The $m = 11$ in the example is prime; since h_2 takes values between 1 and 10, every sequence visits all 11 slots.

14.3.8 Comparing the three probing schemes

The three methods can be written with a single function. The following code gives the function `probe`, which computes the slot of the *i*-th attempt according to the method, and a general insertion function that uses it. The definitions of `table`, `slot` and `M` are the same as in the open addressing section.

```
typedef enum { LINEAR, QUADRATIC, DOUBLE } method_t;

int probe(int k, int i, method_t m) {   /* slot of attempt i */
  int h1 = k % M;
  switch (m) {
    case LINEAR:    return (h1 + i) % M;
    case QUADRATIC: return (h1 + i * i) % M;
    default:        return (h1 + i * (1 + k % 10)) % M;  /* h2 */
  }
}

int insert_m(int k, method_t m) {
  for (int d = 0; d < M; d++) {
    int i = probe(k, d, m);
    if (table[i].state != USED) {
      table[i].key = k;
      table[i].state = USED;
      return i;
    }
  }
  return -1;
}
```

`method_t` names the three methods. The `switch` statement selects one of the formulas according to the method. In the double hashing branch the step size is `1 + k % 10`. Because this code uses 10 rather than *m*, it suits only the table in the example. In a general table h_2 is chosen to depend on *m*, for example *1 + (k mod (m − 2))*. `insert_m` is the same as `insert`; the only difference is that it finds the slot with `probe` instead of `i = (i + 1) % M`. In quadratic probing the product `i * i` does not overflow as long as $i < M$. The test program inserted the six keys separately with each of the three methods; the results are exactly the same as Table 14.8.

Method	Attempt *i*	Probe sequence of 33	Probes	Final table (slot: key)
Linear	$h + i$	0, 1, 2, 3, 4, 5	6	0: 22, 1: 1, 2: 13, 3: 11, 4: 24, 5: 33
Quadratic	$h + i^2$	0, 1, 4, 9	4	0: 22, 1: 1, 2: 13, 3: 24, 4: 11, 9: 33
Double hashing	$h_1 + i \cdot h_2$	0, 4, 8	3	0: 22, 1: 1, 2: 13, 4: 11, 7: 24, 8: 33

Table 14.8: The same table, three probing schemes: where does 33 go? (22, 1, 13, 11, 24 were inserted before it.)

Method	Clustering	Distinct probe sequences
Linear	primary (blocks)	m
Quadratic	secondary (same home address)	m
Double hashing	least	$\approx m^2$

Table 14.9: Type of clustering and number of distinct probe sequences for the three schemes.

Looking at these tables, one might think double hashing should always be chosen. On real computers the situation is more complicated. A processor brings memory into the cache not byte by byte but in blocks of 64 bytes. In linear probing, consecutive slots are in the same block: once the first probe brings the block in, the next few probes are almost free. The jumps of double hashing, in contrast, land in a different block at every probe. Linear probing is therefore cache-friendly and very fast when α is small (for example below 0.5). As α grows, the number of probes rises quickly because of clustering, and double hashing pulls ahead. The choice depends on how full the table will be kept.

14.4 Chaining

In open addressing, a colliding item goes to another slot and occupies the place of other keys. Chaining does the opposite: the colliding item stays in its own bucket. So that a bucket can hold more than one item, every bucket becomes the head of a linked list.

14.4.1 One linked list per bucket

Definition 14.13

Separate chaining, or simply chaining: every bucket of the table is the head pointer of a linked list. All keys whose home address is j are kept in the list of bucket j, that is, in its chain.

```
#define MZ 5
typedef struct node {
  int key;
  struct node *next;
} node;
node *bucket[MZ];           /* all NULL */
```

The definitions are the usual node of a linked list. `struct node` has two fields: the key and the address of the next node. The structure holds a pointer of its own type, and this pointer is written before the `typedef` is complete. That is why the structure is given both a tag name (`struct node`) and a type name (`node`). The array `bucket` consists of five pointers. Because it is global, all pointers are `NULL` when the program starts: all five buckets are empty lists.

The picture in memory differs from open addressing. The array `bucket` itself is small and contiguous: five pointers, 40 bytes on a 64-bit machine. The nodes, however, are obtained one by one with `malloc` and scattered through random places in the heap. On the test machine `sizeof(node)` came out as 16 bytes: a 4-byte key, 4 bytes of padding to align the pointer on an 8-byte boundary, and an 8-byte pointer. For each item, 12 bytes of overhead are added to the 4-byte key. The bookkeeping information that `malloc` keeps for every block is not included in this count either.

```
bool chain_insert(int k) {
  int i = k % MZ;
  node *np = malloc(sizeof *np);
  if (np == NULL) return false;  /* no memory */
  np->key = k;
  np->next = bucket[i];         /* at the head */
  bucket[i] = np;
  return true;
}
```

- **Line 2:** `i` is the bucket of the key.
- **Line 3:** Memory is obtained for the new node. `sizeof *np` is the size of the type that `np` points to, that is, `sizeof(node)`. This spelling stays correct even if the type name changes. `malloc` is in the header `<stdlib.h>`.
- **Line 4:** If `malloc` cannot find memory, it returns `NULL`. If this check is skipped, the next line tries to write to address `NULL` and the program crashes. The function reports the error to the caller by returning `false`.
- **Lines 5–7:** The node is added at the head of the list. First the `next` field of the new node points to the old head, then the bucket points to the new node. The order matters: if `bucket[i] = np` were done first, the old list would be lost.

Insertion at the head takes two pointer assignments however long the list is: *O(1)*. Inserting at the end of the list would require walking the whole list. Since the order within a chain has no meaning, inserting at the head is the cheapest choice. This function too does not prevent the same key from being inserted twice. If duplicates are not wanted, `chain_search` must be called before insertion; then the cost of insertion equals the cost of search.

14.4.2 Code trace: `chain_insert`

The table has five buckets and *h(k) = k mod 5*. Initially bucket 0 holds 10 and bucket 4 holds 4. Then 12, 7 and 22 are inserted in turn; all three have bucket 2. Table 14.10 traces each call.

Call	`i`	`np->next`	`bucket[2]` afterwards	Chain 2
`chain_insert(12)`	2	`NULL` (old head)	node(12)	12
`chain_insert(7)`	2	node(12)	node(7)	7 → 12

Call	i	np->next	bucket[2] afterwards	Chain 2
chain_insert(22)	2	node(7)	node(22)	22 → 7 → 12

Table 14.10: Inserting 12, 7 and 22 with chain_insert (*MZ = 5*).

In the first call bucket 2 is empty. The old head, that is, NULL, is written into the next field of the new node and the bucket points to the new node. Insertion into an empty list needs no special case; the same two assignments give the right result. In the second and third calls the new node goes in front of the old head each time. After the three insertions the chain is 22 → 7 → 12: the most recently inserted is at the head. Each call made one malloc.

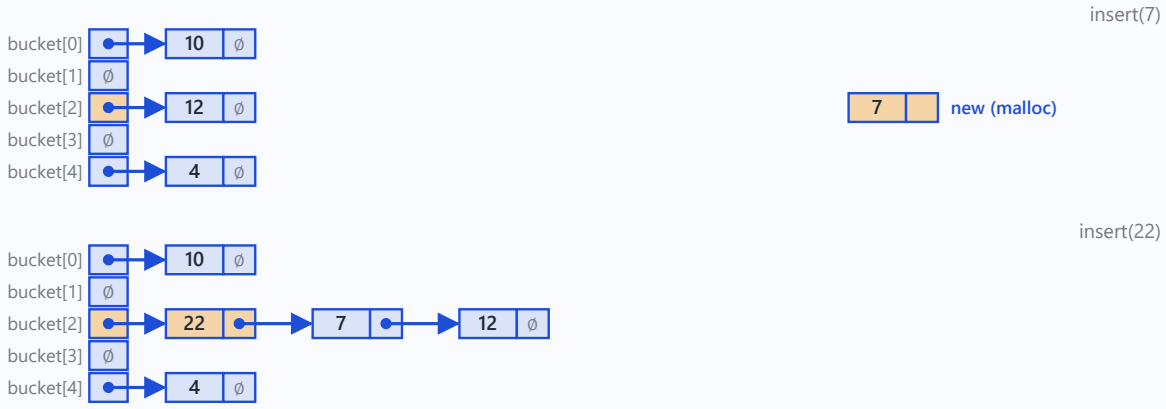


Figure 14.11: Two moments of the trace of chain_insert. Above, chain_insert(7): the next field of the new node points to the old head (12) and the bucket has not been updated yet. Below, after all three insertions: the chain of bucket 2 is 22 → 7 → 12. The Ø sign is NULL.

14.4.3 Code trace: chain_search

```
node *chain_search(int k) {
  node *p = bucket[k % MZ];
  while (p != NULL && p->key != k)
    p = p->next;
  return p;                 /* or NULL */
}
```

Search walks the list of the bucket from the start. p starts at the head of the list. The while condition asks two things: is the list finished, has the key been found? If neither, p moves to the next node. The order of the condition matters. The && operator short-circuits: if p != NULL is false, the second condition is never evaluated. If the order were reversed, NULL->key would be read at the end of the list and the program would crash. When the loop ends, p points either to the node found or to NULL; the function returns both as they are.

Call	Nodes `p` points to	Result	Nodes visited
`chain_search(12)`	22 (≠ 12) → 7 (≠ 12) → 12 (equal)	address of node(12)	3
`chain_search(17)`	22 (≠ 17) → 7 (≠ 17) → 12 (≠ 17) → `NULL`	`NULL`	3

Table 14.11: Trace of `chain_search(12)` and `chain_search(17)`. The chain of bucket 2 is 22 → 7 → 12.

12 is at the end of the chain, because it was inserted first. The search looks at three nodes. The bucket of 17 is also 2 (*17 mod 5 = 2*). All three nodes of the chain differ from 17 and `p` ends up `NULL`. An unsuccessful search must walk the whole chain. An unsuccessful search for a key that falls into bucket 1 or 3, on the other hand, ends without looking at any node: the list is empty.

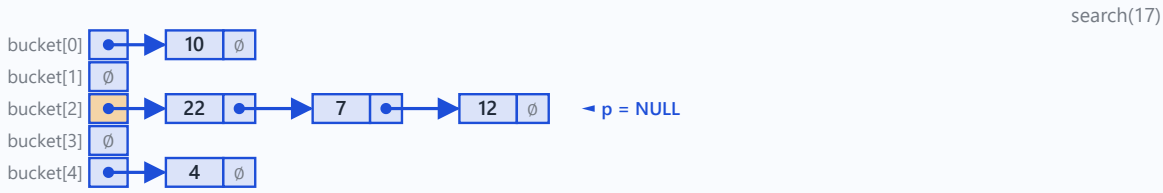


Figure 14.12: The end of the trace of `chain_search(17)`: `p` passed the three nodes of the chain and became `NULL`; 17 is not in the table.

Cost. Search takes the cost of computing the home address plus the number of nodes visited in the chain. Under the simple uniform hashing assumption, the expected length of every chain is $\alpha = n/m$. Because an unsuccessful search walks the whole chain, it looks at α nodes on average; together with the hash computation, $\Theta(1 + \alpha)$. A successful search walks up to the node being sought. The items inserted into the same bucket after the sought item are in front of it, and their expected number is about $\alpha/2$. A successful search therefore looks at $1 + \alpha/2$ nodes on average. In both cases, if α is kept constant the cost is $O(1)$. In the worst case all keys fall into the same bucket and search becomes $O(n)$.

14.4.4 Code trace: `chain_erase`

To delete a node from a singly linked list, the predecessor of the node to be deleted must be known. The `next` field of the predecessor is changed so that it skips the deleted node and points to the one after it. The deletion routine therefore walks with two pointers.

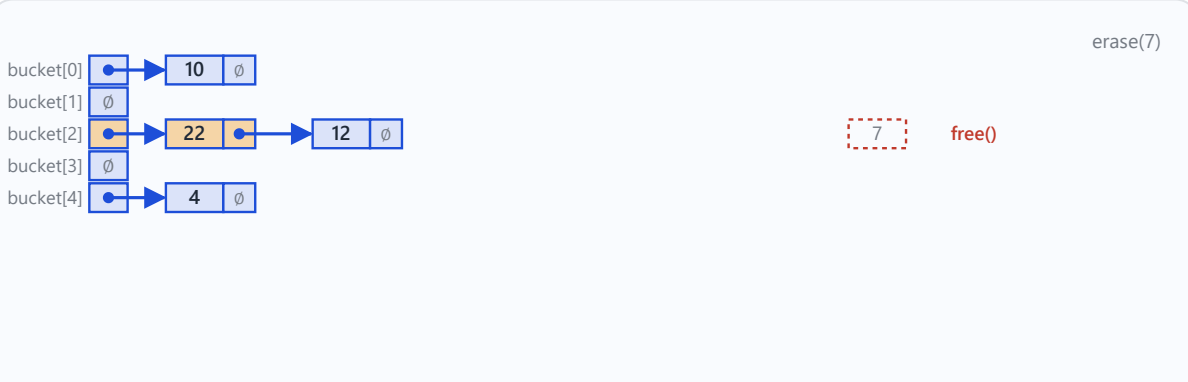
```
bool chain_erase(int k) {
  int i = k % MZ;
  node *p = bucket[i], *prev = NULL;
  while (p != NULL && p->key != k) {
    prev = p;
    p = p->next;
  }
  if (p == NULL) return false;          /* absent */
  if (prev == NULL) bucket[i] = p->next;
  else prev->next = p->next;
  free(p);
  return true;
}
```

- **Line 3:** `p` is the node being examined and `prev` is its predecessor. The node at the head of the list has no predecessor; `prev` therefore starts as `NULL`.
- **Lines 4–7:** The search loop is the same as that of `chain_search`. The only difference is that the old value of `p` is written into `prev` before `p` advances.
- **Line 8:** If `p` is `NULL`, the key is absent and `false` is returned.
- **Lines 9–10:** The node is removed from the chain. If `prev` is `NULL`, the deleted node is the head of the list; the bucket is updated to point to the next node. Otherwise the `next` field of the predecessor points to the successor of `p`.
- **Line 11:** The node's memory is given back with `free`.

Table 14.12 traces the call `chain_erase(7)`; 7 is in the middle of the chain.

Step	`prev`	`p`	Event
1	`NULL`	node(22)	*i = 7 mod 5 = 2*; `p` at the head of the list
2	node(22)	node(7)	22 ≠ 7: `prev` and `p` advance one step
3	node(22)	node(7)	`p->key == 7`: the loop ends
4	node(22)	node(7)	`prev->next = p->next`: 22 now points to 12
5	node(22)	(freed)	`free(p)`; `return true`; chain 22 → 12

Table 14.12: Trace of `chain_erase(7)`. The chain of bucket 2 is 22 → 7 → 12.



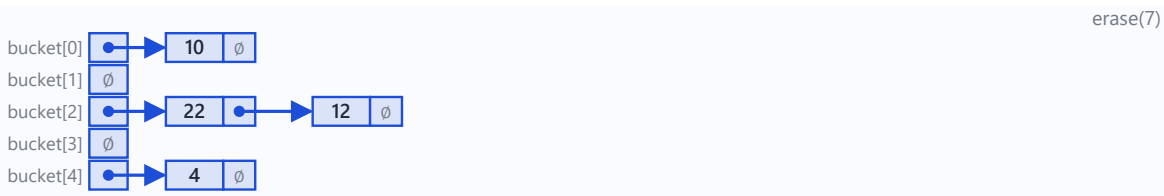


Figure 14.13: `chain_erase(7)`. Above, the node of 7 has been removed from the chain and is about to be freed; 22 points directly to 12. Below, the operation is finished: the chain of bucket 2 is 22 → 12.

After step 4, no pointer from the chain reaches the node of 7. If `free` were not called at this moment, the node would remain in memory but could never be used again: a memory leak. After `free(p)`, however, `p` still holds the old address. That address no longer belongs to the program; `p` is a **dangling pointer**. Reading `p->key` or following `p->next` is undefined behavior. Because the function returns right after `free`, this danger does not arise here. In the test program, a second call `chain_erase(7)` for the deleted 7 returned `false`. Deleting 22 at the head was also tried: `prev` stayed `NULL` and `bucket[2]` pointed to the node of 12.

Deletion in chaining needs no marker as in open addressing. The node is really removed from the list and the search path of no other key changes. Every key is searched only in the list of its own bucket; a deleted node cannot cut the search of another bucket. **Cost:** deletion walks the same path as search and does constant work after finding the node. On average $O(1 + \alpha)$, in the worst case $O(n)$.

Common mistake

Forgetting the deletion of the head node. If the line `prev->next = p->next` runs while `prev` is `NULL`, it writes to address `NULL` and the program crashes. If the node to be deleted is at the head of the list, the pointer that must be updated is not a field of a node but `bucket[i]` itself.

14.4.5 Releasing all the memory of the table

Because the nodes of a chained table are obtained with `malloc`, all of them must be given back before the program ends or when the table is no longer needed. The slides have no such routine; the following code completes it.

```
void chain_clear(void) {
  for (int i = 0; i < MZ; i++) {
    node *p = bucket[i];
    while (p != NULL) {
      node *after = p->next;  /* save before free */
      free(p);
      p = after;
    }
    bucket[i] = NULL;
  }
}
```

The outer loop goes over the buckets and the inner loop over the chain of each bucket. The only subtlety of the code is the variable `after`. After `free(p)`, `p->next` cannot be read, because the node's memory no longer belongs to the program. The address of the next node is therefore saved before `free`. At the end the bucket is set to `NULL` and the table becomes reusable. The cost is *O(n + m)*: each node is handled once and each bucket once.

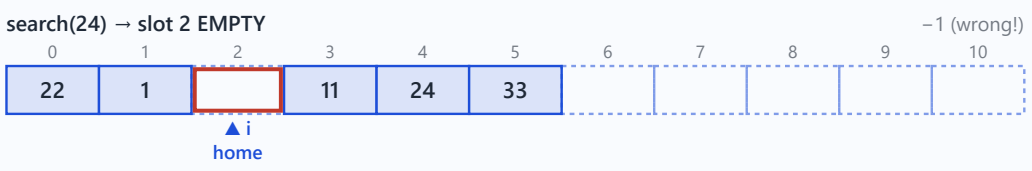
The memory leak was checked with a counting `malloc`/`free` wrapper, because AddressSanitizer cannot be used on Windows. The wrapper increments one counter on every successful allocation and another on every `free`. The test program inserted 10, 4, 12, 7 and 22, deleted 7 and 22, and then called `chain_clear`. The counters showed 5 allocations and 5 releases: there is no leak.

14.5 Deletion, load factor and resizing

Insertion and search in open addressing were built; deletion was postponed. Why deletion needs a section of its own is seen in the first example. The second half of the section deals with how full the table should be kept and how the table is grown once a threshold is passed.

14.5.1 The deletion trap in open addressing

A table with linear probing holds 22, 1, 13, 11, 24 and 33. 13 is to be deleted. The first solution that comes to mind is to set the slot of 13, slot 2, to `EMPTY`. Then let `search(24)` be called. The home address of 24 is 2; 24 had settled in slot 4 because slots 2 and 3 were occupied when it was inserted. The search looks at the home address, finds slot 2 `EMPTY` and returns −1 at once. Yet 24 is in the table.



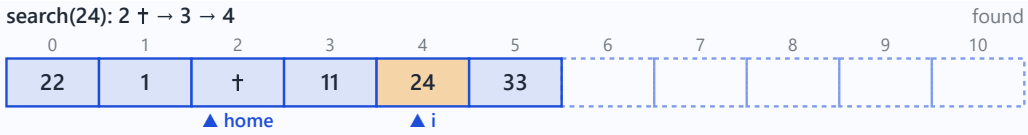


Figure 14.14: Deletion in open addressing. Above, the wrong way: slot 2 was set to `EMPTY` and `search(24)` saw an empty slot at the home address and stopped; 24 was not found. Below, the right way: slot 2 was marked with a deleted marker (†); the search passed this slot, looked at 3 and 4, and found 24.

The source of the problem is the stopping rule of the search. The search stops under the assumption "if the key were present, it would be found before the first `EMPTY` slot". This assumption holds as long as slots only go from `EMPTY` to `USED`. When a `USED` slot becomes `EMPTY` again, the path of every key that settled by passing through that slot is cut. The keys beyond the deleted slot are in the table but cannot be found.

Definition 14.14

A **tombstone** is the special state put on a deleted slot in open addressing (`DELETED` in the code). Search does not count this slot as empty; it passes over it and continues. Insertion does count this slot as empty and may reuse it.

The name means a gravestone: no item lives in the slot any more, but it is recorded that an item once stood there. The tombstone obeys both rules. The path of a search is not cut, because a marked slot is full as far as search is concerned. Space is not wasted, because a marked slot is empty as far as insertion is concerned.

14.5.2 Code trace: `erase(13)`, then `search(24)`

The deletion routine is the second function of the code fragment given in the search section:

```
int search(int k) {
  int i = k % M;
  for (int d = 0; d < M; d++, i = (i + 1) % M) {
    if (table[i].state == EMPTY) return -1;
    if (table[i].state == USED &&
        table[i].key == k) return i;
  }
  return -1;
}
bool erase(int k) {
  int i = search(k);
  if (i < 0) return false;
  table[i].state = DELETED;   /* tombstone */
  return true;
}
```

`erase` first finds the slot of the key with `search`. If the key is absent, it returns `false`. Otherwise it sets the state of the slot to `DELETED`. The key field is not touched; the old value stays in the slot. This causes no problem, because `search` accepts a key only if its state is `USED`. Table 14.13 traces the deletion and the search that follows it.

Call	Step	Event
`erase(13)`	1	`search(13)`: home address *13 mod 11 = 2*; slot 2 is `USED` with key 13 → returns 2 (1 probe)
	2	`i = 2 ≥ 0`: there is an item to delete; `table[2].state = DELETED`
	3	`return true`
`search(24)`	1	home address *24 mod 11 = 2*; slot 2 is `DELETED`: neither `EMPTY` nor `USED`, continue
	2	slot 3: `USED`, key 11 ≠ 24, continue
	3	slot 4: `USED`, key 24 → `return 4` (3 probes)

Table 14.13: Trace of `erase(13)` followed by `search(24)`. Initially: slots 0–5 = 22, 1, 13, 11, 24, 33.

`search(24)` does not stop at the tombstone in slot 2; both `if` conditions are false and the loop advances. The search finds 24 in three probes, as before the deletion. The test program ran the same sequence: `erase(13)` returned 1 (`true`), the following `search(24)` returned 4, `search(13)` returned −1, and a second `erase(13)` returned 0 (`false`). When slot 2 was set to `EMPTY` by hand, `search(24)` returned −1; the trap really occurred.

Insertion and a repeated key. `insert` reuses a `DELETED` slot. This creates a danger that the slides' exercise asks about. With 13 deleted and 24 standing in slot 4, let `insert(24)` be called. `insert` looks at home address 2, accepts the slot because it is not `USED`, and writes 24 there. The table now has two 24s: in slot 2 and in slot 4. In the test program this call really returned 2 and the table became 22, 1, 24, 11, 24, 33. Later `erase(24)` deletes only the first copy; the second copy stays in the table and `search(24)` goes on finding a deleted key. The solution is to search before inserting:

```
int insert_unique(int k) {        /* does not reinsert an existing key */
  int i = search(k);
  if (i >= 0) return i;        /* present: return its slot */
  return insert(k);              /* write to first EMPTY or DELETED slot */
}
```

`insert_unique` scans the entire probe sequence of the key with `search`. If the key is present, it returns its position and writes no new copy. If not, `insert` writes to the first `EMPTY` or `DELETED` slot in the sequence. In the test program, after 13 was deleted, `insert_unique(24)` returned 4 and the table did not change; `insert_unique(46)` wrote 46 into the tombstone slot 2. This version reclaims the space of tombstones and does not allow repeated keys. The price is one search on every insertion.

Tombstones have a cost too. Because marked slots count as full for search, they lengthen searches. In a table with many insertions and deletions, most of the slots may become `DELETED` over time even if the actual number of items is small. In that case unsuccessful searches scan nearly the whole table. The remedy is to rebuild the table at certain intervals: all `USED` items are reinserted into a new, clean table and the markers are discarded. This operation is the same as resizing. In chaining this problem never arises.

14.5.3 Load factor and the expected number of probes

In every collision-resolution scheme, cost rises with how full the table is. The load factor $\alpha = n/m$ measures this fullness. The theoretical analysis of hash tables gives the expected number of probes as a function of α under the simple uniform hashing assumption. Table 14.14 summarizes the results for four values of α.

α	Chaining ✓	Chaining ✗	Linear ✓	Linear ✗	Uniform ✓	Uniform ✗
0.25	1.13	1.25	1.2	1.4	1.15	1.3
0.5	1.25	1.50	1.5	2.5	1.39	2.0
0.75	1.38	1.75	2.5	8.5	1.85	4.0
0.9	1.45	1.90	5.5	50.5	2.56	10.0

Table 14.14: Expected number of probes. ✓ successful, ✗ unsuccessful search. The "Uniform" columns are ideal open addressing, in which the probe sequence of every key is a random permutation.

The numbers in the table come from the following formulas:

- **Chaining:** successful search $1 + \alpha/2$, unsuccessful search $1 + \alpha$ (the first 1 is the access to the bucket). Both are $\Theta(1 + \alpha)$.
- **Linear probing** (Knuth): successful search $\tfrac{1}{2}\,(1 + 1/(1 - \alpha))$, unsuccessful search $\tfrac{1}{2}\,(1 + 1/(1 - \alpha)^2)$.
- **Uniform open addressing** (CLRS): successful search $(1/\alpha) \cdot \ln(1/(1 - \alpha))$, unsuccessful search $1/(1 - \alpha)$. Double hashing behaves very close to this ideal.

There is a simple idea behind the formula $1/(1 - \alpha)$ for an unsuccessful search. Each probe looks at a random slot, and this slot is occupied with probability α. The first probe is always made. The second probe is reached if the first slot is occupied: probability α. The third is reached if the first two slots are occupied: probability about α^2. The expected total is the geometric sum $1 + \alpha + \alpha^2 + \ldots = 1/(1 - \alpha)$. For $\alpha = 0.9$ this is 10: one slot in ten is empty, and finding an empty slot takes ten tries on average.

The second lesson of the formulas is that cost grows not linearly in α but as $1/(1 - \alpha)$. When α goes from 0.5 to 0.75, the unsuccessful search in linear probing goes from 2.5 to 8.5, and at 0.9 to 50.5. In chaining the increase is gentle: at $\alpha = 0.9$ an unsuccessful search is 1.9 probes, and the table keeps working when α exceeds 1.

Because the formulas rest on an assumption, they were checked by measurement. Pseudo-random keys generated with a fixed seed were inserted into a table of 100,003 slots (a prime). For each α, the average number of probes in a successful search was computed over all inserted keys, and in an unsuccessful search over 20,000 keys that are not in the table. In double hashing, $h_2(k) = 1 + (k \bmod (m - 2))$ was used. Table 14.15 gives the results next to the formulas. Since the keys were generated with a fixed seed, the numbers are independent of the machine; only time was not measured.

α	Linear ✓	Linear ✗	Double hashing ✓	Double hashing ✗	Chaining ✓	Chaining ✗ (nodes)
0.25	1.170 (1.167)	1.384 (1.389)	1.154 (1.151)	1.331 (1.333)	1.128 (1.125)	0.244 (0.25)
0.5	1.500 (1.500)	2.499 (2.500)	1.389 (1.386)	2.005 (2.000)	1.248 (1.250)	0.496 (0.50)
0.75	2.521 (2.500)	8.826 (8.500)	1.855 (1.848)	3.991 (4.000)	1.374 (1.375)	0.738 (0.75)
0.9	5.636 (5.500)	51.118 (50.500)	2.551 (2.558)	10.084 (10.000)	1.450 (1.450)	0.907 (0.90)

Table 14.15: Measured average number of probes (formula in parentheses), $m = 100{,}003$. For an unsuccessful search in chaining, the number of nodes visited is given; the 1 in the formula $1 + \alpha$ is the access to the bucket.

The measurements agree with the formulas; the deviations are below 4%. The largest deviations are in linear probing, at $\alpha = 0.75$ and 0.9 (2–4%). At high fullness a few long blocks affect the average noticeably. The values of double hashing match the uniform open addressing formula to three digits.

Theorem 14.2 (Constant cost at a constant load factor)

Under the simple uniform hashing assumption, if the load factor is kept bounded by a constant α_0 less than 1 ($\alpha \le \alpha_0 < 1$), the expected cost of search, insertion and deletion is $O(1)$ in open addressing and in chaining. In chaining, it is enough for α to be bounded by any constant.

Proof. In chaining the expected cost is $\Theta(1 + \alpha)$; if α is bounded by a constant, this is a constant too. In uniform open addressing, an unsuccessful search and insertion take at most $1/(1 - \alpha) \le 1/(1 - \alpha_0)$, and a successful search takes even less. The right-hand side does not depend on n. Deletion is a search plus constant work, so it has the same bound. ■

The thresholds used in practice are as follows: $\alpha \le 0.5$–0.7 in open addressing and $\alpha \le 1$ or so in chaining. In quadratic probing, by the theorem, the threshold must be at most 0.5. When the threshold is exceeded, the table is grown.

14.5.4 Resizing

If the table size is given by a fixed macro, the load factor grows with the number of items. If the number of items is not known in advance, the table's array must be dynamic and must be grown when the threshold is exceeded.

```
typedef struct {
  slot *a;      /* dynamic array of m slots */
  int m, n;     /* size, used slots */
} Table;
/* after inserting: */
if (2 * t->n > t->m)            /* α > 1/2 */
  resize(t, next_prime(2 * t->m));
```

The `Table` structure carries three fields: the address of the slot array, the number of slots `m`, and the number of occupied slots `n`. The array is now obtained with `malloc` or `calloc` and its size can change at run time. The second part of the code is the check that runs after every insertion. The condition `2 * t->n > t->m` means $\alpha > 1/2$; to avoid fractional numbers, both sides were multiplied by $2m$. When the threshold is exceeded, the table is grown to the first prime that is not smaller than twice the old size. A prime size is needed for the division method and for quadratic probing.

The fragment in the slides assumes that two helper functions exist. The versions written for the test program are these:

```
int next_prime(int x) {        /* first prime not smaller than x */
  for (;; x++) {
    int prime = x >= 2;
    for (int d = 2; d * d <= x && prime; d++)
      if (x % d == 0) prime = 0;
    if (prime) return x;
  }
}

int table_insert(Table *t, int k) {  /* linear probing, m = t->m */
  int i = k % t->m;
  for (int d = 0; d < t->m; d++, i = (i + 1) % t->m) {
    if (t->a[i].state != USED) {
      t->a[i].key = k;
      t->a[i].state = USED;
      t->n++;
      return i;
    }
  }
  return -1;
}
```

`next_prime` finds the first prime starting from `x`. It tests each candidate with divisors up to $\sqrt{x}$; the condition `d * d <= x` applies this bound without computing a square root. `table_insert` is the dynamic-table version of `insert` from open addressing. The only differences are that it takes the size from the field `t->m` rather than from the macro `M`, and that it increments the occupied-slot counter. The sequence of sizes that starts at 5 and takes the first prime at or above twice the size each time goes 5, 11, 23, 47, 97, 197, 397, 797, 1597, 3203, ...

Why rehash? The first way that comes to mind to grow the table is to copy the old array to the beginning of the new one. This is wrong. The slot of an item depends on *k mod m*, and when *m* changes the home address changes too. 12 is at *12 mod 5 = 2* in a table of 5 slots; in a table of 11 slots its home address is *12 mod 11 = 1*. A 12 copied to slot 2 cannot be found in the new table by `search(12)`: the search looks at slot 1, finds it empty and stops. Every item must be inserted anew into the new table. This operation is called **rehashing**.

```
bool resize(Table *t, int new_m) {
  slot *old = t->a;
  int old_m = t->m;
  slot *a = calloc(new_m, sizeof *a);  /* EMPTY */
  if (a == NULL) return false;
  t->a = a; t->m = new_m; t->n = 0;
  for (int i = 0; i < old_m; i++)
    if (old[i].state == USED)
      table_insert(t, old[i].key);
  free(old);
  return true;
}
```

- **Lines 2–3:** The address and size of the old array are saved in local variables. Because the table structure will soon be attached to the new array, the old array can be read only through these copies.
- **Lines 4–5:** The new array is obtained with `calloc`. Unlike `malloc`, `calloc` zeroes the memory; since `EMPTY` = 0, all slots start empty. If memory cannot be obtained, the function returns `false`. In that case the table has not changed at all: the old array is in place; it just could not be grown.
- **Line 6:** The table is attached to the new array and `n` is reset; `table_insert` will increment `n` again with each insertion.
- **Lines 7–9:** Every slot of the old array is examined. Only `USED` slots are inserted into the new table. `EMPTY` and `DELETED` slots are skipped; so the tombstones are also cleaned up.
- **Line 10:** The old array is freed with `free`. If this line is forgotten, every growth leaks the old array.

One subtlety of the function is its relationship with `table_insert`. If `table_insert` did the growth check itself, another growth could be triggered during rehashing. Because the new size is more than twice the old size and `n` starts from zero, the threshold is not exceeded in this code. Still, it is safer for the check to stay outside `table_insert`, on the caller's side. That is also the arrangement in the slides.

14.5.5 Code trace: `resize`

7, 12 and 3 have been inserted into a table of 5 slots. With linear probing, 7 settled in slot 2, 12 in slot 3 (home address 2 is occupied), and 3 in slot 4 (home address 3 is occupied). $n = 3$, $m = 5$, $\alpha = 0.6 > 0.5$: the table will be grown. The new size is `next_prime(10)` = 11. Table 14.16 gives the trace.

Step	`i`	Event	`t->n`
1	—	`old` = old array, `old_m` = 5	3
2	—	`calloc(11, …)`: 11 slots, all `EMPTY`; `t->a` is the new array, `t->m = 11`	0
3	0, 1	`old[0]`, `old[1]` are `EMPTY`: skipped	0
4	2	7 is `USED`: *7 mod 11 = 7* → slot 7 (old home address *7 mod 5 = 2*)	1

Step	i	Event	t->n
5	3	12 is `USED` : *12 mod 11 = 1* → slot 1 (old home address 2)	2
6	4	3 is `USED` : *3 mod 11 = 3* → slot 3 (old home address 3)	3
7	—	`free(old)` ; `return true` ; new $\alpha = 3/11 \approx 0.27$	3

Table 14.16: Trace of `resize(t, 11)`. Old table: slots 2, 3, 4 = 7, 12, 3.

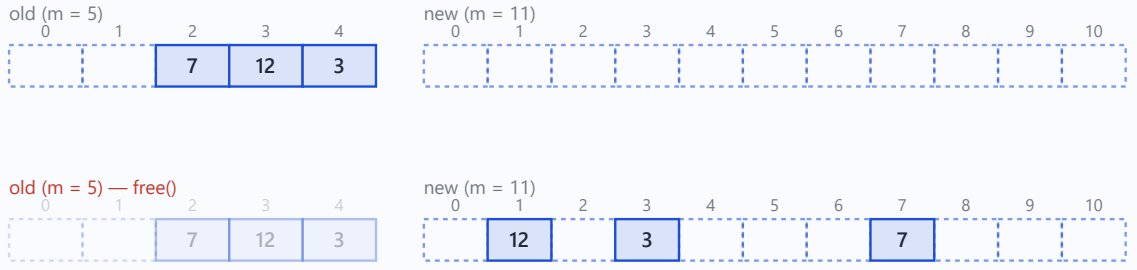


Figure 14.15: `resize(t, 11)`. Above, `calloc` has produced 11 empty slots; 7, 12 and 3 are in the old table. Below, the three items have settled at their new home addresses (12 → 1, 3 → 3, 7 → 7) and the old array has been freed.

In the old table the three items formed a contiguous block, and 12 and 3 had slipped away from their home addresses. In the new table all three are at their own home addresses; there is no collision. Besides lowering the load factor, the growth has also dispersed the clusters. The test program got the same result in two ways: when `resize` was called directly, and when 7, 12, 3 were inserted in turn with the threshold check. In the second way the growth was triggered right after the third insertion. The counting wrapper showed four allocations (two initial arrays and two new arrays) and four releases.

Cost of a single growth. While `calloc` zeroes the new array it touches m' slots, the loop goes over the old m slots, and n items are reinserted. The total is $O(n + m)$. A single insertion is therefore sometimes very expensive.

Theorem 14.3 (Amortized cost of growth)

If the table is grown by doubling its size (or more) when the threshold is exceeded, then over n insertions into an empty table the total number of reinserted items is less than $2n$. The amortized cost per insertion is $O(1)$.

Proof. Let the last growth have taken place when the number of items was $n_s \le n$, and let n_s items be reinserted in this growth. The previous growth was triggered at the latest when the table size was half of this; at that moment there were at most $n_s/2$ items. In the one before that, at most $n_s/4$, and so

on. The total number of reinsertions is at most $n_s + n_s/2 + n_s/4 + \ldots < 2n_s \le 2n$. The `calloc` cost of each growth is also proportional to the table size, and since the sizes at least double each time, it is bounded by the same geometric sum. The total cost of n insertions is $O(n)$ and $O(1)$ per insertion. ■

This kind of analysis is called **amortized analysis**: although some individual operations are expensive, the total cost of a sequence of operations is divided by the number of operations. The theorem was checked by a measurement. One million items were inserted into a table starting at 5 slots, and when $\alpha > 1/2$ the table was grown to the next prime. There were 19 growths and the final size reached 3,292,489. The total number of reinsertions was 1,646,169, that is, 1.65 per item: below the 2 in the theorem. The total number of slots allocated in the growths came to 6.6 per item.

> **Common mistake**
>
> Growing the table by a fixed amount each time, for example 100 slots. Then n insertions cause about $n/50$ growths, and each moves as many items as there are at that moment, n. The total work reaches the order of n^2 and the amortized cost per insertion becomes $O(n)$. Growth must be multiplicative.

14.6 Chaining or open addressing?

The two collision-resolution schemes offer the same dictionary operations, and both are $O(1)$ on average when α is constant. The choice between them depends on the properties of the data and the application. Table 14.17 puts the differences side by side.

Criterion	Chaining	Open addressing
Load factor	$\alpha > 1$ is possible; the table does not fill up, it slows down	$\alpha \le 1$; cost explodes as $\alpha \to 1$
Deletion	simple; the node is removed from the list	needs a tombstone; tombstones accumulate
Bad hash function	more robust; only the affected chain grows	clusters affect neighboring keys too
Memory	a pointer per item and `malloc` overhead	no extra pointers; the whole table is contiguous
Cache	nodes scattered in the heap; not cache-friendly	contiguous array; linear probing in particular is cache-friendly
Choice of probing	not needed	a good probing scheme (double hashing) must be chosen against clustering

Table 14.17: Comparison of chaining and open addressing.

The greatest strength of chaining is its flexibility. Even if the number of items is not known in advance, the table keeps working; only the chains grow. Deletion is as simple as removing a node from a linked list and does not affect the rest of the table. A bad hash function lengthens some chains,

but the damage stays limited to those buckets. The price is memory and cache. On the test machine a 16-byte node was allocated for a 4-byte key. Because nodes are in different places of the heap, every step along a chain may require the processor to fetch a new block of memory.

Open addressing is superior on these two points. All items are in a single array, there are no pointers, and `malloc` is called only when the table is grown. The consecutive probes of linear probing fall into the same cache block. The price is sensitivity to the load factor and the complexity of deletion. If α is kept around 0.5, half of the table stays empty; compared with the node overhead of chaining, this is often still cheap.

A practical rule can be summarized as follows. If the number of items cannot be predicted, if there are many deletions, or if the quality of the hash function is uncertain, choose chaining. If memory and speed are critical, deletions are rare, and the load factor can be kept low, choose open addressing. Real libraries use both. Java's `HashMap` is based on chaining; Python's `dict` is based on open addressing.

14.7 Complexity summary

Table 14.18 collects the cost of the hash table operations. The average column assumes that the load factor is kept constant and that the hash function spreads keys uniformly.

Operation	Average (α constant)	Worst	Comment
search	*O(1)*	*O(n)*	if all keys fall into the same bucket
insert	*O(1)*	*O(n)*	insertion at the head of a chain is always *O(1)*
delete	*O(1)*	*O(n)*	tombstone in open addressing
resize	*O(n + m)*	*O(n + m)*	amortized *O(1)* per insertion
sorted traversal	*O(m + n lg n)*	*O(m + n lg n)*	a hash keeps no order; items are collected and sorted
minimum / maximum	*O(n + m)*	*O(n + m)*	the whole table is scanned

Table 14.18: Complexity of hash table operations (n items, m buckets).

The last two rows are where a hash table falls behind a balanced search tree. A search tree finds the smallest item in *O(lg n)* and traverses the items in sorted order in *O(n)*. In a hash table the items are scattered over the buckets, so all buckets are examined to find the smallest item. For sorted traversal, all items are first collected into an array and then sorted.

The worst case does not come only through bad luck; it can also be produced deliberately. If the hash function is known to everyone, an attacker can prepare thousands of keys that all fall into the same bucket. For example, if a web server keeps request parameters in a hash table, a single request full of parameter names that fall into the same bucket puts the server to $O(n^2)$ work. This attack is called **hash flooding**. The remedy is to choose the hash function at random when the program starts. The function is chosen at random from a predefined family. Because the attacker does not know which function was selected, they cannot prepare colliding keys. If the probability that any two distinct keys

collide is at most *1/m* for the family, the family is called **universal**. This method is called **universal hashing** (CLRS, Section 11.3.3). The string hashes of languages such as Python and Rust therefore start with a random key on every run.

14.8 Common mistakes

The six most frequent mistakes in hash table code are the following. Each is a situation encountered with an example in these notes.

1. `k % m` **coming out negative for a negative key.** In C, `-7 % 11` is −7. A negative index writes outside the array. The key should be kept `unsigned`, or the remainder corrected as in `h_safe`.
2. **Choosing *m* as 10 or a power of 2.** With the division method, this choice looks only at the last digits or the last bits. Multiples of 10 collect in a single bucket at *m = 10*. *m* should be a prime far from powers of 2.
3. **Setting a deleted slot to** `EMPTY` **in open addressing.** Searches for keys that settled by passing through the deleted slot stop halfway. The slot must be marked `DELETED`.
4. **Falling into an infinite loop when the table is full.** If the probe loop runs until it finds an empty slot, it never stops in a full table. The number of attempts must be limited to *m*. In quadratic probing not all slots are probed even in *m* attempts; the table must not be filled beyond half.
5. **Not checking the result of** `malloc`**; using a node after** `free`**.** Writing to the result of a `malloc` that returned `NULL` crashes the program. Reading the `next` field of a deleted node after `free` is undefined behavior; the address must be saved first.
6. **Copying items with their old indices during resizing.** The home address depends on *m*. An item copied to its old position cannot be found in the new table. Every item must be reinserted into the new table.

14.9 Exercises and solutions

The solutions of the six exercises of the slides are given below with their intermediate steps. All results were recomputed with a verification script.

14.9.1 A small table with linear probing

Question. With *m = 7*, *h(k) = k mod 7* and linear probing, what does the table become when 10, 17, 24, 3 are inserted in turn?

Solution. The home addresses of the four keys are *10 mod 7 = 3*, *17 mod 7 = 3*, *24 mod 7 = 3* and *3 mod 7 = 3*: all are synonyms. 10 is written in slot 3. For 17, slot 3 is occupied, so it is written in slot 4. For 24, slots 3 and 4 are occupied, so it is written in slot 5. For 3, slots 3, 4 and 5 are occupied, so it is written in slot 6. The final table: slots 3–6 = 10, 17, 24, 3. The numbers of probes are 1, 2, 3 and 4; 10 in total.

14.9.2 The same keys with chaining

Question. If the same keys are placed with chaining (insertion at the head), how many nodes does the longest chain have?

Solution. All four keys fall into bucket 3. Since insertion at the head puts the last inserted at the front, the chain becomes 3 → 24 → 17 → 10: four nodes. The other six buckets are empty. The hash function has not spread these four keys at all; *k mod 7* is a bad choice for numbers that are 3 more than a multiple of 7. Chaining confines this bad distribution to a single bucket, but a search in that bucket still takes four nodes.

14.9.3 A quadratic probe sequence

Question. With *m = 11*, write the probe sequence of quadratic probing from 0, 1, 4, 9, 16, 25 (mod 11).

Solution. The first four values are smaller than 11 and stay as they are: 0, 1, 4, 9. *16 mod 11 = 5* and *25 mod 11 = 3*. The sequence is 0, 1, 4, 9, 5, 3. All six values are distinct; this agrees with the bound *⌊11/2⌋ = 6* in the quadratic probing theorem. The seventh attempt is *36 mod 11 = 3*, and it starts to repeat.

14.9.4 A failed search in a full table

Question. In a table with linear probing and *α = 0.9*, how many probes does an unsuccessful search expect?

Solution. By Knuth's formula, $\tfrac{1}{2}\left(1 + 1/(1 - 0.9)^2\right) = \tfrac{1}{2}\left(1 + 1/0.01\right) = \tfrac{1}{2} \cdot 101 = 50.5$ probes. In the measurement 51.1 was found. For double hashing at the same fullness the expected value is 10, and for chaining 1.9. As *α → 1*, the cost of linear probing explodes.

14.9.5 Additive and polynomial hashes

Question. Compute the additive hash and the base-31 polynomial hash (mod 11) for “ab” and “ba”.

Solution. “a” = 97, “b” = 98. Additive: *97 + 98 = 195* and *195 mod 11 = 8*; both strings fall into bucket 8. Polynomial: for “ab”, *97 · 31 + 98 = 3105* and *3105 mod 11 = 3*; for “ba”, *98 · 31 + 97 = 3135* and *3135 mod 11 = 0*. The polynomial hash separates the two anagrams. `str_hash` gives the same values by taking the mod at intermediate steps: for “ab”, first *97 mod 11 = 9*, then *(9 · 31 + 98) mod 11 = 377 mod 11 = 3*.

14.9.6 Searching before inserting

Question. In open addressing, why must `insert` call `search` before writing into a `DELETED` slot?

Solution. The same key may already be present in a slot after the tombstone. In the example of the deletion section, after 13 is deleted, 24 stands in slot 4; the first slot in the probe sequence of 24 is the tombstone slot 2. An unchecked `insert(24)` writes 24 a second time into slot 2. If `search` is called

first, 24 is found in slot 4 and no insertion is made. The function `insert_unique` does this.

14.10 Summary

- A hash table does not compare keys; it computes the position from the key. Search, insertion and deletion are *O(1)* on average and *O(n)* in the worst case. The price is the loss of order information: sorted traversal, minimum/maximum and range queries are not cheap.
- Direct addressing uses the key as the index; it cannot be built if the universe is large. A hash function $h: U \to \{0, \ldots, m-1\}$ reduces the key to a small range. Because $m < |U|$, collisions are unavoidable; the birthday paradox shows that the first collision is expected after about $\sqrt{m}$ keys.
- A good hash function is fast, spreads uniformly, uses all the bits and is deterministic. In the division method, m is chosen as a prime far from powers of 2. In the multiplication method, $A \approx 0.618$ and $m = 2^p$ can be used. Long keys are folded; strings are computed with a base-31 polynomial hash by Horner's rule. In C, the negative result of `%` must be watched for.
- In open addressing all items are in the table and $\alpha \le 1$. Linear probing creates primary clustering and quadratic probing creates secondary clustering; double hashing minimizes clustering with a key-specific step size. In quadratic probing m must be prime and $\alpha \le 1/2$; in double hashing h_2 must not be zero and must be relatively prime to m.
- In open addressing a deleted slot is not set to `EMPTY` but marked with a tombstone. Search passes over the marker, and insertion reuses the marked slot; to prevent repeated keys, a search is made before insertion.
- In chaining every bucket is the head of a linked list. Insertion at the head is *O(1)* and search is $\Theta(1 + \alpha)$. Deletion is removing a node from a linked list; the predecessor pointer and the head-node case need care. All nodes must finally be freed with `free`.
- The expected number of probes grows on the order of $1/(1-\alpha)$. When the threshold is exceeded, the table is grown to the first prime above twice its size and all items are rehashed; the amortized cost is *O(1)* per insertion.
- The worst case can be triggered by malicious input; universal hashing protects against this by choosing the function at random at run time.

14.11 Key terms

hash table hash function direct addressing bucket collision load factor dictionary abstract data type division method multiplication method folding open addressing probe linear probing quadratic probing double hashing primary clustering separate chaining tombstone rehashing amortized analysis cache dangling pointer memory leak

14.12 Test Yourself

1. With *m = 11*, *h(k) = k mod 11* and linear probing, 20, 31, 42, 9, 53 are inserted in turn. What is the final table? In how many probes does 53 settle?

   Answer

   All five keys have home address 9. 20 → 9, 31 → 10, 42 → 9, 10 occupied, wrap around → 0, 9 → 9, 10, 0 occupied → 1, 53 → 9, 10, 0, 1 occupied → 2. Table: slot 0 = 42, 1 = 9, 2 = 53, 9 = 20, 10 = 31. 53 settles in five probes.

2. The same keys are put into a chained table with *m = 11* buckets, inserting at the head. What does the chain of bucket 9 look like? How many nodes does `chain_search(20)` look at?

   Cevap

   The chain is 53 → 9 → 42 → 31 → 20; the last inserted is at the head. Since 20 is at the end of the chain, the search looks at five nodes.

3. The keys 16, 32, 48 and 64 are put through the division method with *m = 16* and *m = 17*. What are the home addresses? What does the result show?

   Cevap

   With *m = 16* all four fall into 0, because *k mod 16* takes only the last four bits and the last four bits of these keys are zero. With *m = 17* the home addresses are 16, 15, 14 and 13; all four are different. A table size that is a power of 2 creates clustering on patterned data; a prime size does not.

4. What does `str_hash("ali", 7)` return? (a = 97, l = 108, i = 105)

   Cevap

   Round 1: *97 mod 7 = 6*. Round 2: *6 · 31 + 108 = 294*, *294 mod 7 = 0*. Round 3: *0 · 31 + 105 = 105*, *105 mod 7 = 0*. The result is 0.

5. In a table with linear probing, slots 0–5 = 22, 1, 13, 11, 24, 33. `erase(22)` is called. How many probes does the following `search(33)` make? What would have happened if slot 0 had been set to

`EMPTY`?

Cevap

Slot 0 becomes `DELETED`. `search(33)` passes over the marker in slot 0, finds different keys in 1, 2, 3 and 4, and finds 33 in slot 5: 6 probes. If slot 0 had been set to `EMPTY`, the search would have stopped at the first probe and returned −1; 33 would not have been found although it is in the table.

6. In a chained table *n = 2000*, *m = 1000*; in an open-addressing table *α = 0.8* and linear probing is used. How many probes does a successful and an unsuccessful search take on average in each?

Cevap

In chaining *α = 2*: successful *1 + α/2 = 2*, unsuccessful *1 + α = 3*. In linear probing, successful *½ (1 + 1/0.2) = 3*, unsuccessful *½ (1 + 1/0.04) = 13*. The chained table is faster although it is twice as full.

7. Keys are inserted in turn into an empty table of 11 slots; after each insertion, if `2 * n > m`, the table is grown to size `next_prime(2 * m)`. After which insertion do the first two growths happen, and what are the new sizes?

Cevap

The first growth happens after the 6th insertion (*12 > 11*); the new size is `next_prime(22)` = 23. The second happens after the 12th insertion (*24 > 23*); the new size is `next_prime(46)` = 47.

8. Double hashing is used with *m = 10* and for one key $h_1 = 5$, $h_2 = 6$. What is the probe sequence and why is it problematic?

Cevap

The sequence is *(5 + 6i) mod 10*: 5, 1, 7, 3, 9, 5, 1, … Only five slots are probed, because 6 and 10 have a common divisor of 2. Even if the table has room, this key may fail to find it. If *m* is chosen prime, the problem disappears.

9. The chain of bucket 2 is 22 → 7 → 12 and `chain_erase(22)` is called. How many rounds does the loop run, and which pointer is updated?

Cevap

The loop does not run at all: `p` is the first node and its key is 22. `prev` stays `NULL`; therefore `bucket[2] = p->next` is executed and the bucket now points to the node of 7. Then the node of 22 is freed; the chain becomes 7 → 12.

10. The key is `int k = -15` and $m = 11$. What is `k % m` in C? What is the correct home address?

Cevap

In C, `-15 % 11` is −4; this index is outside the array. The correct home address is $-4 + 11 = 7$ (`h_safe(-15, 11)`).

14.13 Further reading

- E. Horowitz, S. Sahni and S. Anderson-Freed, *Fundamentals of Data Structures in C* (2nd ed.), Chapter 8: the source followed in these notes for static and dynamic hashing, the division, mid-square and folding methods, and linear and quadratic probing with chaining.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms* (3rd ed.), Chapter 11: gives direct addressing, the analysis of chaining, the division and multiplication methods, universal hashing and proofs of the expected number of probes in open addressing.
- R. Sedgewick, *Algorithms in C, Parts 1–4* (3rd ed.), Chapter 14: treats hash functions, chaining, linear probing and double hashing with C code and experimental comparisons.
- D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching* (2nd ed.), Section 6.4: contains the original analysis of the formulas $\frac{1}{2}\left(1 + 1/(1-\alpha)\right)$ and $\frac{1}{2}\left(1 + 1/(1-\alpha)^2\right)$ for linear probing and the rationale for the choice of the golden ratio in the multiplication method.

14.14 References

1. E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2nd ed., Chapter 8 (Hashing).
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 3rd ed., MIT Press, Chapter 11 (Hash Tables).
3. R. Sedgewick, *Algorithms in C, Parts 1–4*, 3rd ed., Addison-Wesley, Chapter 14 (Hashing).
4. D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 2nd ed., Addison-Wesley, 1998, Section 6.4.

PART IV · CHAPTER 15

Graphs

A graph is the most general data structure for holding objects together with the connections between them; linked lists and trees are special cases. This chapter opens with Euler's Königsberg bridge problem. It then defines the basic notions of vertex, edge, path, degree and connectivity, and proves the degree-sum theorem. The core of the chapter is how to store a graph in memory: the adjacency matrix, the adjacency list and its reverse, a sequential representation in a single array, the orthogonal list and the multilist are all built in C, and their memory and operation costs are compared. Next comes the basic operation on a graph, traversal. Depth-first search is written recursively and breadth-first search with a dynamic queue, and both are traced step by step. The last part uses the two traversals to find connected components, to look for cycles and to test whether a graph is bipartite. Every routine comes with a line-by-line walkthrough, a trace table and a complexity analysis.

LEARNING OUTCOMES

Define
Use G = (V, E), directed and undirected graph, adjacency, subgraph, path, degree and connectivity correctly.

Prove
Justify that the degrees sum to 2e and that a connected graph has at least n − 1 edges.

Implement
Build the adjacency matrix, adjacency list, sequential array, orthogonal list and multilist in C, and draw how each looks in memory.

Traverse
Code depth-first (DFS) and breadth-first (BFS) search, trace them by hand and analyze them as O(n + e).

Apply
Find connected components, cycles and bipartiteness with DFS or BFS.

Choose
Pick the right representation by looking at the operations the problem needs: data structure first, code second.

15.1 Introduction: bridges, maps and networks

The data structures so far imposed a particular shape on the relationships between items. In an array or a linked list every item has at most one predecessor and one successor. In a tree every node has a single parent, and children never point back up. Many real-world relationships fit neither pattern. Many roads leave a city and tangle with one another before meeting again in other cities. A computer in a network connects to several others, and the connections can form loops. The structure that records such relationships without imposing any restriction is called a **graph**.

15.1.1 The seven bridges of Königsberg

The history of graph theory is usually dated to 1736. The city of Königsberg in East Prussia lay on both banks of the river Pregel, and two islands sat in the middle of the river. In the sketch on the left of Figure 15.1, the islands are labeled A and D and the banks B (south) and C (north). Seven bridges connected these four pieces of land: two from A to C, two from A to B, one from A to D, one from D to C and one from D to B.

The townspeople asked: can one start from any piece of land, cross every bridge *exactly once* and return to the starting point? Trial and error produced no answer. Leonhard Euler (pronounced “Oiler”) put aside the details of the map to settle the question. He drew each piece of land as a point and each bridge as a line joining two points. Today the points are called **vertices** (or nodes) and the lines **edges**. The resulting drawing is the graph on the right of the figure. There are two separate edges between A and C, because there are two separate bridges.

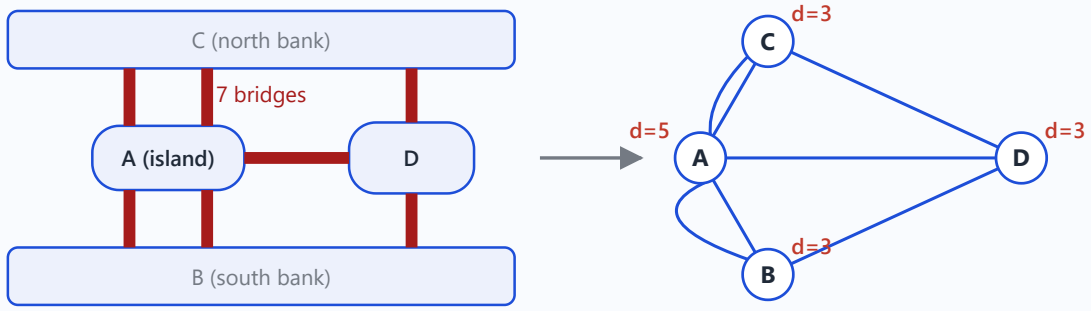


Figure 15.1: The Königsberg bridges (left) and the graph derived from them (right). The red strips are bridges. The degree of each vertex is written next to it: A has degree 5, the other three have degree 3.

The importance of this abstraction is the following. What decides the question is not the size of the land masses, the length of the bridges or the bends of the river. All that matters is which piece connects to which, and by how many bridges. The position of a vertex on the paper is irrelevant as well: vertex A may be drawn to the left or to the right. The same graph can be drawn in many different ways. A drawing is only a picture; the graph itself consists of nothing but the connections.

Euler tied the question to a number. The number of edges touching a vertex is the **degree** of that vertex. Every time the walk enters an intermediate vertex, it must also leave it, and the entry and exit must use different bridges. So every visit uses up two of the vertex's edges. The starting vertex is in the same situation: the walk leaves it and finally enters it again. In a closed walk that uses every bridge once, the edges at every vertex must therefore be used up in pairs. Such a walk can exist only if *every vertex has even degree*. In Königsberg the degrees are 5, 3, 3 and 3, all four odd. The walk does not exist.

Euler's result still carries his name. A closed walk that traverses every edge of a connected graph exactly once is called an **Euler circuit**. Such a circuit exists if and only if all degrees are even. Necessity follows from the counting argument above; the proof of sufficiency is longer and belongs to a graph theory course. What matters in this chapter is that a question about a city map can be answered purely in terms of numbers and connections.

15.1.2 Where graphs are used

Since Königsberg, graphs have entered almost every field of engineering. Their use is so widespread that it is harder to find a field that does not use graphs than to list the ones that do. Table 15.1 collects the main examples.

Area	Vertex	Edge	Typical question
Electric circuits	Junction	Circuit element	What are the currents and voltages?
Project planning (activity networks)	Event or task	Precedence relation	When is the earliest the project can finish?
Chemistry	Atom	Bond	Do two molecules have the same structure?
Road and computer networks	City, router	Road, link	Which path is the shortest?
Transport networks	Depot, pump	Pipe (with capacity)	How much flow can pass from the source to the sink at most?
Infrastructure design	Village	Possible cable route	How can all villages be connected with the least cable?
Formal languages	State	Transition	Does the finite automaton accept this word?
Network reliability	Computer	Link	Is the network disconnected, and into how many pieces?

Table 15.1: Main areas in which graphs are used.

Some of the questions in the table are classic problems known by name. The shortest path problem asks for the cheapest route between two cities on a map. The **maximum flow** problem asks how much oil or water a pipe network can carry per unit time. The **minimum spanning tree** problem asks to join all villages with the smallest total cable length. Trees are themselves graphs; topological sorting and finite automata are also defined on graphs. All of these problems rest on the same foundation: how to store a graph in memory and how to move around in it. This chapter builds exactly that foundation.

15.2 Definitions and terminology

Every routine that deals with graphs relies on a few basic notions. This section defines them precisely and introduces the example graphs used in the rest of the chapter.

15.2.1 Definition of a graph

> **Definition 15.1**
>
> A **graph** $G = (V, E)$ consists of two finite sets. $V(G)$ is a nonempty set of **vertices**. $E(G)$ is a possibly empty set of **edges**, each a pair of vertices. The number of vertices is written $n = |V|$ and the number of edges $e = |E|$.

Requiring the vertex set to be nonempty is a necessity: no operation on a graph without vertices would mean anything. The edge set, however, may be empty. A graph made of a single vertex, or of vertices that are not connected to one another at all, is also valid. In the sparse graph example below, 7 of the 11 vertices touch no edge.

If the order of an edge's two endpoints matters, the graph is **directed** (a digraph); if not, it is **undirected**. The notation shows the difference through the shape of the brackets:

- In an undirected graph an edge is an unordered pair and is written with ordinary parentheses: $(v0, v1)$ and $(v1, v0)$ are the same edge.
- In a directed graph an edge is an ordered pair and is written with angle brackets: $\langle v0, v1\rangle \neq \langle v1, v0\rangle$. The first vertex, where the arrow starts, is called the **tail**; the second, where the arrow points, is the **head**: $\langle tail, head\rangle$.

The edge set is a subset of the Cartesian product $V \times V$ of the vertex set with itself. This product contains all ordered pairs; a graph selects only some of them. The selected pairs say which vertex is connected to which.

Four small example graphs are used throughout the chapter. Figure 15.2 draws them, and Table 15.2 lists their vertex and edge sets.

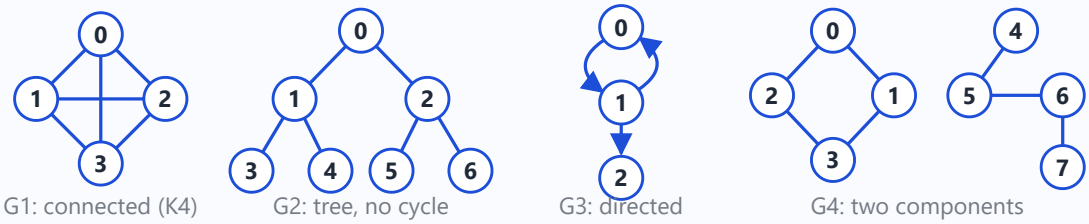


Figure 15.2: Example graphs. G1 is the complete graph on four vertices. G2 is a tree with seven vertices. G3 is a directed graph with three vertices; there is one edge in each direction between 0 and 1. G4 has two parts: {0, 1, 2, 3} and {4, 5, 6, 7}.

Graph	V(G)	E(G)	n	e
G1	{0, 1, 2, 3}	{(0,1), (0,2), (0,3), (1,2), (1,3), (2,3)}	4	6
G2	{0, 1, ..., 6}	{(0,1), (0,2), (1,3), (1,4), (2,5), (2,6)}	7	6
G3	{0, 1, 2}	{<0,1>, <1,0>, <1,2>}	3	3
G4	{0, 1, ..., 7}	{(0,1), (0,2), (1,3), (2,3), (4,5), (5,6), (6,7)}	8	7

Table 15.2: Vertex and edge sets of the example graphs.

In the edge set of G3, *<0,1>* and *<1,0>* are written separately. In a directed graph these are two different edges; the presence of one does not require the other. In an undirected graph it is enough to write *(0,1)*; *(1,0)* is just another way of writing the same edge.

Unless stated otherwise, the graphs in this chapter are **simple**: there is at most one edge between two vertices, and no edge connects a vertex to itself. An edge that connects a vertex to itself is called a **self-loop**. The Königsberg graph is not simple, since there are two edges between A and C. Such graphs are called multigraphs.

Common mistake

Assuming that *<v, u>* exists whenever *<u, v>* is given in a directed graph. In G3, *<1,2>* exists but *<2,1>* does not: nothing can be reached from 2. In an undirected graph every edge can be used in both directions; in a directed graph, only in the direction of the arrow.

15.2.2 Adjacent vertices and subgraphs

Definition 15.2

If *(v0, v1)* is an edge of an undirected graph, *v0* and *v1* are **adjacent**, and the edge is **incident** on these two vertices. In a directed graph, if *<v0, v1>* exists, then *v0* is adjacent to *v1*, and *v1* is adjacent from *v0*.

In G1 every vertex is adjacent to the other three. In G2 the neighbors of 0 are 1 and 2; 3 and 4 are not adjacent to each other, because there is no edge between them. Both are adjacent to 1, but that does not make them adjacent to each other.

Definition 15.3

A graph G' is a **subgraph** of G if $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$.

The definition has a hidden condition: G' must be a graph in its own right. That is, both endpoints of every edge in $E(G')$ must lie in $V(G')$. An edge with an unselected endpoint cannot enter the subgraph. Since the edge set may be empty, a single vertex is also a valid subgraph.

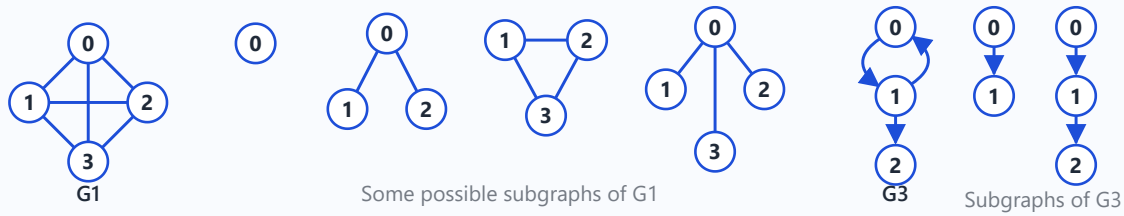


Figure 15.3: On the left, G1 and four of its subgraphs: the single vertex 0; vertices 0, 1, 2 with two edges; the triangle 1, 2, 3; and three edges attached to 0. On the right, G3 and two of its subgraphs: only *<0,1>*; and *<0,1>* with *<1,2>*.

The number of subgraphs of a graph grows quickly. For G1 alone there are $2^6 = 64$ ways to choose the edges, and the choices of vertices add to that. The figure shows only a few of them.

15.2.3 Paths, connectivity and degree

Definition 15.4

A **path** from vertex v_p to vertex v_q is a sequence of vertices $v_p, v_{i1}, \ldots, v_{in}, v_q$ such that each of the pairs $(v_p, v_{i1}), (v_{i1}, v_{i2}), \ldots, (v_{in}, v_q)$ is in $E(G)$. The **length** of a path is the number of edges it contains. In a directed graph the pairs must have the form $\langle \cdot, \cdot \rangle$, that is, follow the direction of the arrows.

In G2, the sequence 3, 1, 0, 2, 5 is a path of length 4 from 3 to 5. A path whose vertices are all distinct is a **simple path**. A path with the same first and last vertex, no other repeated vertex and at least one edge is a **cycle**. In G1, 0, 1, 2, 0 is a cycle. Since G2 is a tree, it contains no cycle at all. In an undirected simple graph a cycle has at least three edges; the sequence 0, 1, 0 goes along an edge and comes back along the same edge, so it does not count as a cycle.

Definition 15.5

In an undirected graph, two vertices are **connected** if there is a path from *v0* to *v1*. A graph is itself connected if every pair of distinct vertices (v_i, v_j) is connected.

G1 and G2 are connected. G4 is not connected: there is no path from 0 to 5. G4 consists of two separate parts. These parts are called connected components; components are treated in detail in the last section of this chapter.

Definition 15.6

The **degree** *d(v)* of a vertex is the number of edges that have it as an endpoint. In a directed graph, the number of edges coming into a vertex is its **in-degree** and the number of edges leaving it is its **out-degree**. The degree of the vertex is the sum of the two.

Among undirected examples, *d(0) = 2* in G2, *d(1) = 3* in G1 and *d(4) = 1* in G2. In the directed graph G3, the two numbers are computed separately for each vertex (Table 15.3).

Vertex	Incoming edges	In	Outgoing edges	Out	Degree
0	<1,0>	1	<0,1>	1	1 + 1 = 2
1	<0,1>	1	<1,0>, <1,2>	2	1 + 2 = 3
2	<1,2>	1	—	0	1 + 0 = 1

Table 15.3: In-degrees and out-degrees in G3.

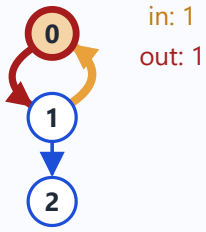


Figure 15.4: Vertex 0 in G3. The light arrow is the edge coming into 0 (in-degree 1), and the dark red arrow is the edge leaving 0 (out-degree 1). The degree of vertex 0 is *1 + 1 = 2*.

Degree is always stated for a particular vertex. A sentence such as “the degree of G3 is 2” is meaningless; the correct statement is “the degree of vertex 0 in G3 is 2”. What can be said about a graph as a whole is its maximum degree or the sum of its degrees.

15.2.4 The sum of the degrees

There is a simple but much-used relation between degrees and the number of edges.

Theorem 15.1 (Degree sum)

In a graph G with n vertices and e edges, if d_i denotes the degree of the i-th vertex, then

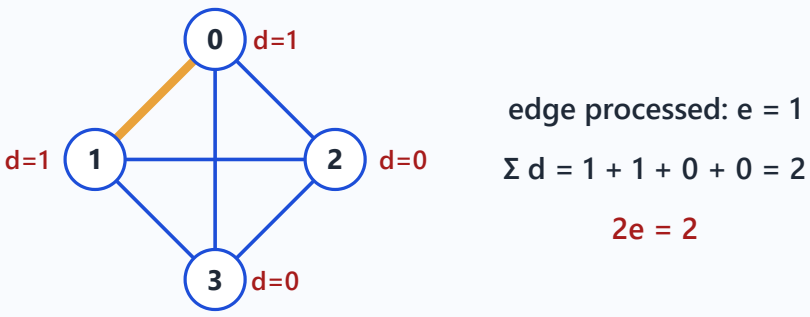
$$d_0 + d_1 + \ldots + d_{n-1} = 2e$$

holds.

Proof. The sum of the degrees is obtained by counting, at each vertex, the edges that touch it and adding up these counts. How many times is each edge counted in this sum? An edge *(u, v)* is counted once in the degree of *u* and once in the degree of *v*. It contributes to no other vertex's degree. So every edge makes exactly 2 contributions to the sum, and the total is *2e*. The same argument works for a directed graph: *<u, v>* contributes one to the out-degree of *u* and one to the in-degree of *v*. ■

The proof counts edges one by one, so having several edges between two vertices changes nothing. The theorem therefore also holds for multigraphs. In the Königsberg graph, $5 + 3 + 3 + 3 = 14 = 2 \cdot 7$. If self-loops are allowed, such an edge is counted as contributing 2 to the degree of its vertex, and the theorem still holds.

Figure 15.5 shows the theorem step by step on G1. Edges are added one at a time to an empty graph. Each addition raises the degrees of the two endpoints by one, so the total grows by 2. After all six edges are added, the total reaches 12.



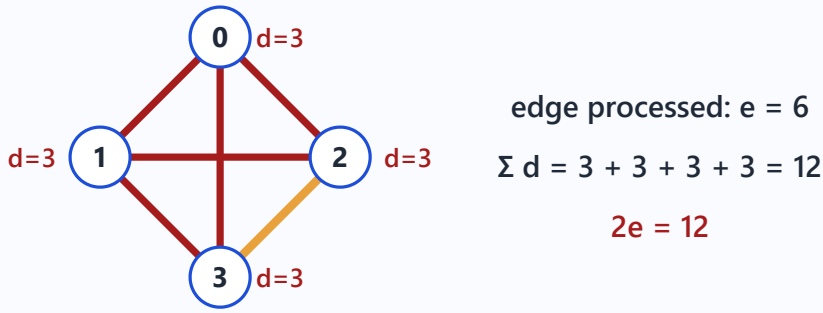


Figure 15.5: The degree-sum theorem. Top: the first edge (0,1) has been added: $\Sigma d = 2 = 2e$. Bottom: all six edges of G1 have been added: every vertex has degree 3, and the total is $12 = 2 \cdot 6$.

The theorem has an immediate corollary: in any graph, the number of vertices of odd degree is even. Since the total $2e$ is even, the number of odd terms cannot be odd. This gives a quick first answer to the question “is there a graph with these degrees?”.

Example 15.1

Is there a simple undirected graph with degree sequence (3, 3, 2, 2, 1)? Add the degrees: $3 + 3 + 2 + 2 + 1 = 11$. The sum is odd, but by the theorem it must equal $2e$, an even number. No such graph exists, and the question about the number of edges loses its meaning. Had the sequence been (3, 3, 2, 2, 2), the sum would be 12 and the number of edges 6. In that case the theorem gives only a necessary condition; whether the graph really exists must be shown separately by construction. For example, if the edge (0,2) is added to the cycle 0–1–2–3–4–0, the degrees are (3, 2, 3, 2, 2), and renaming the vertices gives the sequence.

The theorem matters for data structures as well. In the adjacency list of an undirected graph every edge appears twice, once in the list of one endpoint and once in the list of the other. The total number of items in the lists is the sum of the degrees, that is, $2e$. A routine that scans all the lists therefore costs $O(n + e)$. This is a line of reasoning that returns in almost every analysis below.

15.3 Implementing a graph in memory

Concepts are understood by drawing them on paper, but a program cannot see a drawing. To a program, a graph is nothing but numbers and pointers in memory. The subject of this section is not discrete mathematics; it is the question “how do we store a graph in memory?” There are many representations of the same graph, and the choice among them determines the speed of the operations we will perform on it. Five representations are covered here:

1. the adjacency matrix,
2. the adjacency list and its inverse,
3. a sequential representation in a single array,

4. the orthogonal list for directed graphs, and
5. the multilist.

On top of these comes the weighted representation, which attaches a cost to each edge. Three questions are asked of every representation: How much memory does it need? How fast can the question "is there an edge between u and v?" be answered? How fast can all the neighbors of a vertex be found? The speed of algorithms such as traversal depends most on this last question.

15.3.1 The adjacency matrix

Definition 15.7

The **adjacency matrix** of a graph G with n vertices is a two-dimensional $n \times n$ array a. We have $a[i][j] = 1$ if $(i, j) \in E(G)$ in an undirected graph, or $\langle i, j \rangle \in E(G)$ in a directed graph, and $a[i][j] = 0$ otherwise.

In C the matrix is stored in an integer array sized for the maximum number of vertices: `int a[MAX_V][MAX_V];`. Table 15.4 shows the matrices of G1 and G3 side by side.

	0	**1**	**2**	**3**
0	0	1	1	1
1	1	0	1	1
2	1	1	0	1
3	1	1	1	0

G3	**0**	**1**	**2**
0	0	1	0
1	1	0	1
2	0	0	0

Table 15.4: The adjacency matrix of G1 (left) and of G3 (right). The row gives the tail of an edge and the column its head.

Since $(0,2)$ is an edge of G1, both $a[0][2]$ and $a[2][0]$ are 1. Every edge of an undirected graph is written into the matrix twice, in two cells symmetric about the diagonal. So the matrix of an undirected graph is **symmetric**: $a[i][j] = a[j][i]$ for all i, j. In a simple graph there are no self-loops, so the diagonal is all zeros. The matrix of G3, on the other hand, is not symmetric: $a[1][2] = 1$ but $a[2][1] = 0$.

Symmetry can be used to save space. If only the lower or upper triangle of the matrix is stored, the number of cells roughly halves. But memory is still proportional to $n(n - 1)/2$ cells, so asymptotically it stays $O(n^2)$. The memory of the matrix does not depend on the number of edges at all: a graph with no edges and a complete graph take the same space. A 50-vertex `int` matrix takes 10,000 bytes with 4-byte `int`s.

Two operations are very easy with a matrix. Whether the edge *(u, v)* exists is found in *O(1)* by looking at a single cell. Adding or deleting an edge is just writing 1 or 0 into one cell, again *O(1)*. The other operations require scanning rows:

- **Degree.** In an undirected graph the degree of *i* is the number of 1s in row *i* (or, equivalently, in column *i*). Since a row has *n* cells, the cost is *O(n)*. In a directed graph the row sum gives the out-degree and the column sum the in-degree; together they read *2n* cells, and the cost is again *O(n)*. In G3 the out-degree of 2 is 0 (row 2 is empty) and its in-degree is 1 (there is one 1 in column 2).
- **Number of edges.** Count all the 1s and divide by two (Theorem 15.1). G1 has twelve 1s, so it has 6 edges. By symmetry it is enough to look at the upper triangle, but that still means *n(n − 1)/2* cells. The cost never drops below $O(n^2)$.
- **Neighbors of a vertex.** The *n* cells of the row are scanned: *O(n)*. The whole row is read even if the vertex has a single neighbor.

The last item determines the cost of traversal algorithms and is the real weakness of the matrix. In a sparse graph most rows are mostly zeros, but there is no way to skip the zeros.

The matrix of G4 and a symmetry check

The most frequent mistake when writing a matrix by hand is to record an edge in one direction only. Figure 15.6 shows G4, and Table 15.5 gives its correct matrix.

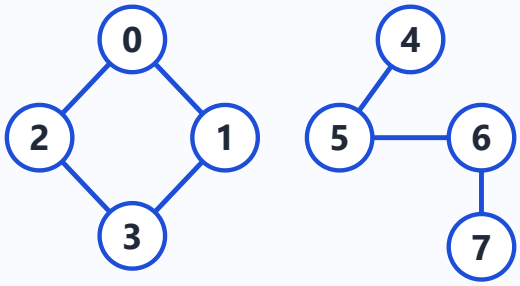


Figure 15.6: The graph G4. It has two components: the four-cycle on vertices 0, 1, 2, 3 and the chain 4–5–6–7.

	0	1	2	3	4	5	6	7
0	0	1	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0
2	1	0	0	1	0	0	0	0
3	0	1	1	0	0	0	0	0
4	0	0	0	0	0	1	0	0
5	0	0	0	0	1	0	1	0
6	0	0	0	0	0	1	0	1

	0	1	2	3	4	5	6	7
7	0	0	0	0	0	0	1	0

Table 15.5: The adjacency matrix of G4. The matrix is symmetric; the two components form two separate blocks on the diagonal.

Common mistake

Writing row 1 of G4 as `1 0 1 0 0 0 0 0` and row 7 as `… 0 0 1`. The extra 1 in the first row adds the nonexistent edge (1,2), and the 1 in the seventh row connects 7 to itself. Both break symmetry: *a[1][2] = 1* while *a[2][1] = 0*. The correct row 1 is `1 0 0 1 0 0 0 0` (neighbors 0 and 3), and the correct row 7 is `0 0 0 0 0 0 1 0` (the only neighbor is 6). Checking the written matrix with the comparison *a[i][j] == a[j][i]* catches this error immediately.

The matrix can also be used to find connected components. Starting from one vertex, mark every vertex that can be reached; then restart from the first unmarked vertex. But finding the neighbors of each vertex is a row scan, that is $O(n)$, so the total cost is $O(n^2)$. The same job is much cheaper with an adjacency list.

Degree from the matrix: code and trace

The routine that computes the degree of a vertex in a directed graph sums the row and the column separately:

```
int degree(int a[][MAX_V], int n, int v)
{
    int j, out = 0, in = 0;
    for (j = 0; j < n; j++)
        out += a[v][j];      /* row */
    for (j = 0; j < n; j++)
        in += a[j][v];      /* column */
    return out + in;
}
```

- **Line 1.** When a two-dimensional array is passed as a parameter, the second dimension must be written: `int a[][MAX_V]`. The compiler computes the address of *a[v][j]* as $v \cdot MAX_V + j$ and so must know the row length. The first dimension may be left empty. `n` is the actual number of vertices; only the upper-left $n \times n$ part of the array is used.
- **Line 3.** Both counters are zeroed. If the initial value is forgotten, local variables start with indeterminate values and the result is random.
- **Lines 4–5.** Row `v` is scanned: `a[v][j]` is 1 if the edge *<v, j>* exists. The sum is the out-degree.

- **Lines 6–7.** Column `v` is scanned: `a[j][v]` is 1 if the edge *<j, v>* exists. The sum is the in-degree.
- **Line 8.** The sum of the two is returned.

If the routine is applied to an undirected graph, row and column are the same, so every edge is counted twice and twice the degree is returned. In an undirected graph the first loop alone is enough. Table 15.6 traces the call `degree(a, 3, 1)` on the matrix of G3.

Step	Line	`j`	Cell read	`out`	`in`
1	3	—	—	0	0
2	4–5	0	a[1][0] = 1	1	0
3	4–5	1	a[1][1] = 0	1	0
4	4–5	2	a[1][2] = 1	2	0
5	6–7	0	a[0][1] = 1	2	1
6	6–7	1	a[1][1] = 0	2	1
7	6–7	2	a[2][1] = 0	2	1
8	8	—	return 2 + 1	result 3	

Table 15.6: Trace of `degree(a, 3, 1)` (G3). Line numbers refer to the code above.

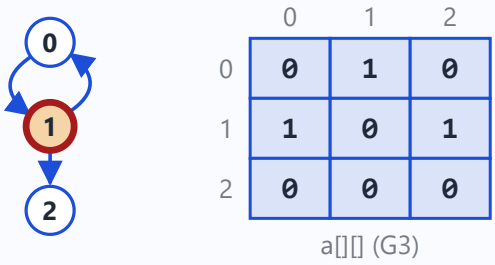


Figure 15.7: G3 and its matrix at the end of the trace. Vertex 1 is highlighted; the two 1s in row 1 give the out-degree and the single 1 in column 1 gives the in-degree.

Each loop runs exactly *n* times, and *2n* cells are read in total. The cost is *O(2n) = O(n)* and is independent of the vertex's actual degree. The same routine returns 2 for vertex 0 and 1 for vertex 2 of G3; these values agree with Table 15.3.

15.3.2 The adjacency list

Each row of the matrix is mostly full of zeros. The adjacency list does not store those zeros at all. Instead of each row of the matrix there is a linked list, and the list holds only the numbers of the columns that contain a 1.

Definition 15.8

In the **adjacency list** representation there is a linked list for each vertex *i*; the items of the list are the numbers of the neighbors of *i*. The head pointers of the lists are kept in an array `graph` of *n* entries: `graph[i]` points to the first item of the neighbor list of *i*.

The C declaration uses a self-referential structure:

```
#define MAX_V 50   /* max number of vertices */
typedef struct node *node_ptr;
typedef struct node {
    int vertex;         /* the neighbor's number */
    node_ptr link;      /* next neighbor in list */
} node;                 /* don't forget ';' and the name */
node_ptr graph[MAX_V];   /* one list per vertex */
```

- **Line 1.** `MAX_V` is the largest number of vertices the program can hold. The array sizes are determined by this constant.
- **Line 2.** `node_ptr` is the pointer type that points to `struct node`, whose definition has not been written yet. In C a pointer to a structure can be declared before the structure is defined; the size of the structure need not be known. This order makes it easy to use a field that points to the structure's own type inside the structure.
- **Lines 3–6.** Each list item carries two fields: the neighbor's number `vertex` and `link`, which points to the next item in the list. The definition ends with `;` after the closing `}` and the type name `node`. If the semicolon is forgotten, the compiler usually reports the error on the following line, and the location of the error looks surprising.
- **Line 7.** The `graph` array reserves a list head for each vertex. Since it is a global array, all its elements are `NULL` when the program starts; that is, every list is initially empty.

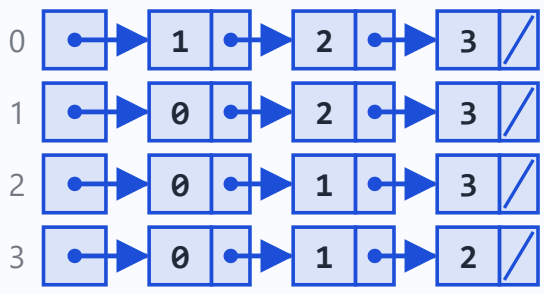


Figure 15.8: The adjacency list of G1. The boxes on the left are the cells of the `graph` array. Each item has two compartments: the neighbor's number and the link to the next item. A slash marks a `NULL` link.

In Figure 15.8 every vertex has three neighbors and every list has three items. There are 12 items in total, while G1 has 6 edges. In an undirected graph every edge appears once in the list of each of its endpoints. This is a disadvantage of the representation: the same information is stored in two places, and deleting an edge requires updating both lists.

The memory and time costs of lists are as follows:

- **Memory.** n heads and, in an undirected graph, $2e$ items: $O(n + e)$. On x86-64 a `node` takes 16 bytes (a 4-byte `int`, 4 bytes of padding and an 8-byte pointer); a head takes 8 bytes.
- **Degree.** In an undirected graph the items in the list of i are counted: $O(d_i)$. Compared with $O(n)$ in the matrix, this is a big gain when the degree is small.
- **Number of edges.** The items in all lists are counted and divided by two: one must look at n heads and count $2e$ items, for a total of $O(n + e)$. This is far less than $O(n^2)$ in the matrix, because in most graphs e is much smaller than n^2.
- **Is (u, v) present?** v is searched for in the list of u: $O(d_u)$. This question was $O(1)$ in the matrix; it is the only place where the list loses.

This is why the complexity of graph algorithms is usually written in terms of both n and e. The expression "$O(n + e)$" describes an algorithm that touches each vertex and each edge a constant number of times.

Undirected and directed examples

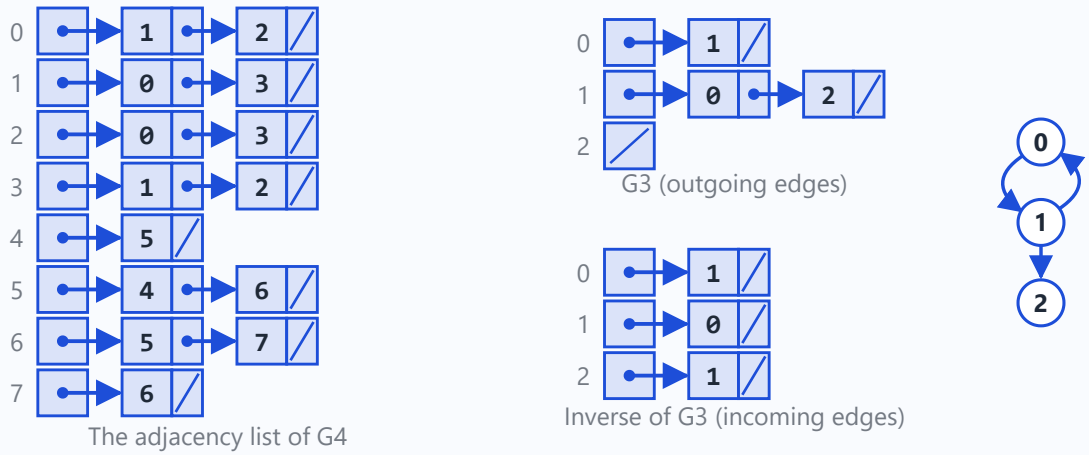


Figure 15.9: On the left the adjacency list of G4; on the right the list of G3 (outgoing edges) and its inverse (incoming edges). The lists are sorted in increasing order.

If the lists of the undirected graph G4 are kept sorted, counting the edges gets a little shorter. Every edge (i, j) appears in the list of both i and j. If only items with $j > i$ are counted, every edge is counted once and there is no need to divide by two. In a sorted list the items with $j < i$ are gathered at the front, for example the 1 and 2 in the list of 3. These items are skipped quickly and the rest are counted. The number of scanned items decreases, but the asymptotic cost is still $O(n + e)$.

The situation is different in the directed graph G3. Since *<0,1>* and *<1,0>* are separate edges, every item is an edge and all of them must be counted. The list holds only outgoing edges: the number of items in `graph[i]` is the out-degree of *i* and is found in *O(d)*. The in-degree, however, is not written in the list directly. To find the edges coming into *i*, one must walk through all the lists and count the items that contain *i*: *O(n + e)*. If the in-degree is asked for often, this cost is unacceptable. The solution is to keep the reverse of the list as well, that is, a list of incoming edges. This inverse adjacency list is built in the next subsection.

A sparse graph: G5

The difference between the matrix and the list is best shown by a sparse graph. G5 has 11 vertices and only 2 edges: *(1, 4)* and *(7, 8)*.

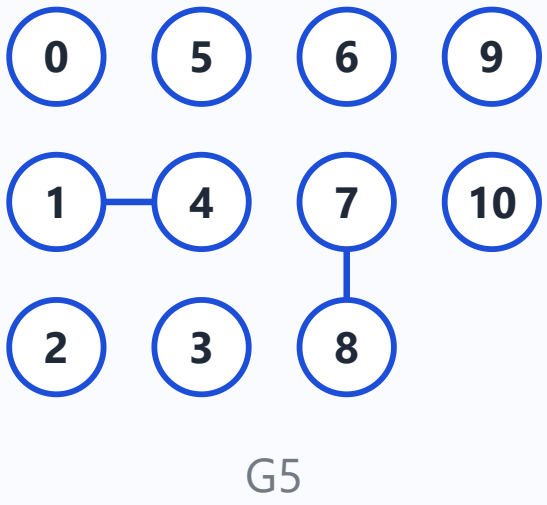


Figure 15.10: G5: 11 vertices, 2 edges. The seven vertices that touch no edge are components on their own; the graph has 9 components in total.

The adjacency list holds four items: `graph[1]` → 4, `graph[4]` → 1, `graph[7]` → 8 and `graph[8]` → 7. The remaining seven heads are `NULL`. In total 11 heads and *2e = 4* items are stored. The matrix, on the other hand, needs *11 × 11 = 121* cells; only 4 of them are 1, and 117 cells are wasted.

A general rule follows: for a sparse graph (*e* small compared with n^2) consider the list, and for a dense graph (*e* close to n^2) the matrix. In a dense graph the pointers of the list take much more room than the bits of the matrix. But the decision is not made on density alone. The real criterion is the operations the program will perform on the graph. If the program constantly asks “are these two vertices adjacent?”, the matrix can be faster even for a sparse graph. If the program walks through the neighbors of every vertex, the list can be the better fit even for a dense graph.

15.3.3 Building the list: adding an edge

In the examples, a person drew the lists by looking at the graph. A program cannot look at the graph. The edges reach it one at a time from a file or from the user, and each must be added to the structure the moment it arrives. The routine that does this is:

```
void add_edge(int u, int v)   /* u -> v */
{
    node_ptr p = malloc(sizeof(node));
    if (p == NULL) exit(1);       /* out of memory */
    p->vertex = v;
    p->link = graph[u];   /* at front: O(1) */
    graph[u] = p;
}
```

- **Line 1.** The routine adds a single edge in the direction $u \to v$: it puts v into the list of u. For an undirected edge *(u, v)* the routine is called twice, as `add_edge(u, v)` and `add_edge(v, u)`.
- **Line 3.** Heap memory is allocated for the new item. The `void *` returned by `malloc` can be assigned to `node_ptr` in C without a cast. Writing `sizeof(node_ptr)` instead of `sizeof(node)` is a common mistake: only the size of a pointer (8 bytes) is allocated, whereas the item is 16 bytes, and later assignments write into unallocated memory.
- **Line 4.** If `malloc` cannot find memory, it returns `NULL`. Without the check, the next line writes through a `NULL` pointer and the program crashes. The simplest reaction is chosen here: the program terminates with an error code.
- **Line 5.** The neighbor's number is written into the item.
- **Line 6.** The link of the new item points to the current first item of the list. If the list is empty this value is `NULL`, and the new item is also the last item of the list.
- **Line 7.** The list head points to the new item. The order of lines 6 and 7 matters: if line 7 were written first, the old list head would be lost and the rest of the list would be unreachable.

The routine does not scan the list; every edge is placed in a constant number of steps, *O(1)*. An undirected graph with e edges is built with $2e$ calls; with the zeroing of the heads included, construction takes $O(n + e)$.

Table 15.7 traces how the lists form when the edges arrive in the order (0,1), (0,2), (1,2). In the table, `&node(k)` denotes the address of the item whose neighbor number is k.

Call	p->vertex	p->link	graph[0]	graph[1]	graph[2]
add_edge(0, 1)	1	NULL	1	—	—
add_edge(1, 0)	0	NULL	1	0	—
add_edge(0, 2)	2	&node(1)	2 → 1	0	—
add_edge(2, 0)	0	NULL	2 → 1	0	0
add_edge(1, 2)	2	&node(0)	2 → 1	2 → 0	0

Call	p->vertex	p->link	graph[0]	graph[1]	graph[2]
add_edge(2, 1)	1	&node(0)	2 → 1	2 → 0	1 → 0

Table 15.7: Building the adjacency list of a triangle with three edges. Each undirected edge is added with two calls.

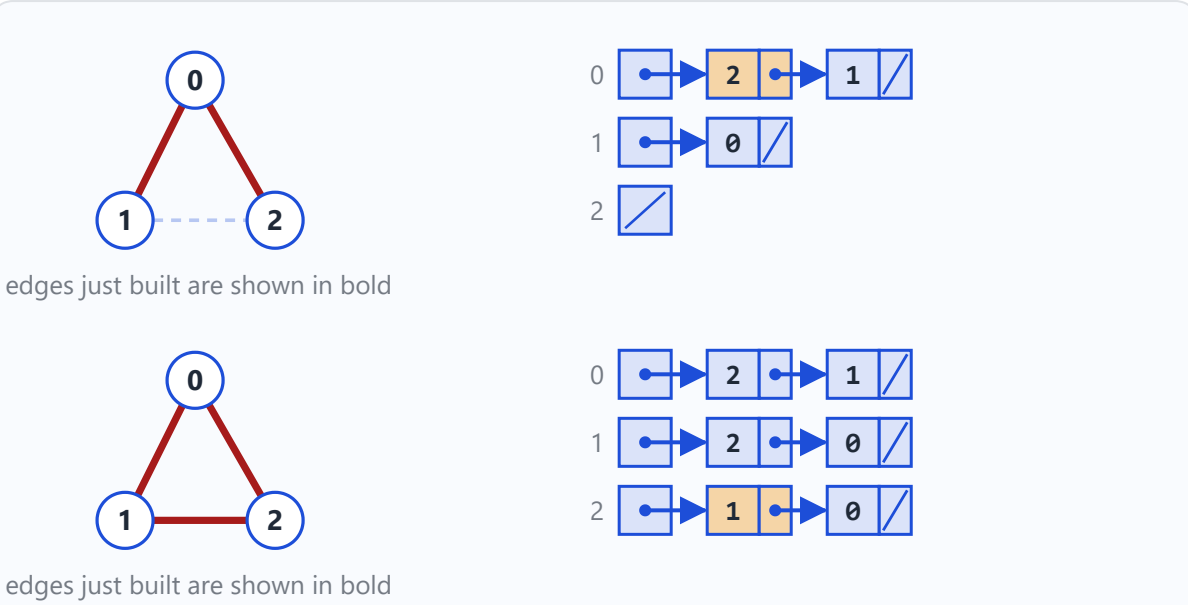


Figure 15.11: Building the list. Top: after the first three calls, with 2 and 1 in `graph[0]` and 0 in `graph[1]`. Bottom: after all six calls. Since every new item enters at the front of its list, the lists are in the reverse of the arrival order.

In the end the list of 0 is 2 → 1, although the edges arrived in the order 1, 2. Insertion at the front reverses the list relative to the arrival order. This is not a bug, but it affects the output of traversal algorithms, because a traversal visits neighbors in the order of the list. That a change in list order changes the traversal order is clearly visible in the examples in the traversal section.

Inserting at the front, at the end or in sorted order

An item can enter the list in three places. The choice depends on the desired order and cost (Table 15.8).

Method	One insertion	List order	Note
Insert at front	O(1)	Reverse of arrival order	Construction O(n + e)
Insert at end (walking)	O(d)	Arrival order	Walk to `NULL` every time
Insert at end (with a tail pointer)	O(1)	Arrival order	Keep an extra `last[i]` for every list
Sorted insertion	O(d)	Ascending	Traversal order becomes predictable

Table 15.8: Options for inserting into an adjacency list. *d* is the current length of the list.

Sorted lists do not depend on the order in which the graph was built; the traversal output depends only on the graph. This is why the lists in the trace examples of this chapter are sorted. Sorted insertion is written compactly with the pointer-to-pointer technique familiar from linked lists:

```
void sorted_insert(int u, int v)   /* v into list u, ascending */
{
    node_ptr *pp = &graph[u];   /* pointer to pointer */
    node_ptr p;
    while (*pp && (*pp)->vertex < v)
        pp = &(*pp)->link;
    p = malloc(sizeof(node));
    if (p == NULL) exit(1);
    p->vertex = v;
    p->link = *pp;
    *pp = p;
}
```

`pp` holds the address of the pointer field to be modified. Initially this field is `graph[u]`. While the loop skips items smaller than *v*, it advances `pp` to the `link` field of each item. When the loop ends, `*pp` points to the item in front of which the new item will go (or to `NULL`). The new item is written into the place of `*pp`. Insertion at the head, in the middle and at the end is done with the same two lines; special cases such as “is the list empty, or are we inserting at the head?” are not needed.

Giving the memory back

Every `malloc` call must be matched by a `free` call. At the end of the program, or when the graph is no longer needed, every list `graph[i]` is walked and its items are released:

```
void free_lists(node_ptr adj[], int n)
{
    int i;
    node_ptr p;
    for (i = 0; i < n; i++) {
        while (adj[i]) {
            p = adj[i];
            adj[i] = p->link;   /* save link first */
            free(p);
        }
    }
}
```

The order inside the loop is the familiar rule for linked lists: the link is saved before the item is released. Reading `p->link` after `free(p)` accesses released memory and is undefined behavior. The routine touches each item once: $O(n + e)$. All the C code in this chapter was run with a counting `malloc` / `free` wrapper; when the lists were released with this routine, the numbers of allocated and freed blocks were equal.

Common mistake

Not checking the result of `malloc` while building the list, and not freeing the lists at the end of the program. In a small test program these mistakes are invisible. In a long-running program that builds and tears down graphs again and again, the memory leak accumulates slowly, and eventually `malloc` really does return `NULL`.

15.3.4 The inverse adjacency list

In a directed graph the adjacency list holds the outgoing edges. The inverse list holds the incoming edges: `rev[j]` is the list of vertices that send an edge to *j*. This is the adjacency list of the **transpose** graph, in which all the arrows are reversed. The inverse list is built by walking the existing lists once:

```
void transpose(node_ptr adj[], node_ptr rev[],
               int n)
{
    int i;
    node_ptr p;
    for (i = 0; i < n; i++) rev[i] = NULL;
    for (i = 0; i < n; i++)
        for (p = adj[i]; p; p = p->link)
            insert(rev, p->vertex, i);  /* <j,i> */
}
```

The `insert` used by the routine is the form of `add_edge` that works on any array of heads:

```
void insert(node_ptr adj[], int u, int v)   /* v into list adj[u] */
{
    node_ptr p = malloc(sizeof(node));
    if (p == NULL) exit(1);
    p->vertex = v;
    p->link = adj[u];
    adj[u] = p;
}
```

- **Line 6.** All heads of the inverse list are set to `NULL`.
- **Lines 7–8.** The list of each vertex *i* is walked from start to end. An item numbered *j* in the list means the edge *<i, j>*.
- **Line 9.** In the transpose graph this edge becomes *<j, i>*: *i* is added to the list `rev[j]`. The comment `<j,i>` says exactly this.

Line	i	p->vertex	Edge	Added
6	—	—	—	`rev[0..2] = NULL`

Line	`i`	`p->vertex`	Edge	Added
8–9	0	1	<0,1>	0 to `rev[1]`
8–9	1	0	<1,0>	1 to `rev[0]`
8–9	1	2	<1,2>	1 to `rev[2]`
7–8	2	—	—	`adj[2]` empty, nothing to add
7	3	—	—	loop ends

Table 15.9: Trace of `transpose(adj, rev, 3)` (G3). `adj[0]` = 1, `adj[1]` = 0 → 2, `adj[2]` = NULL.

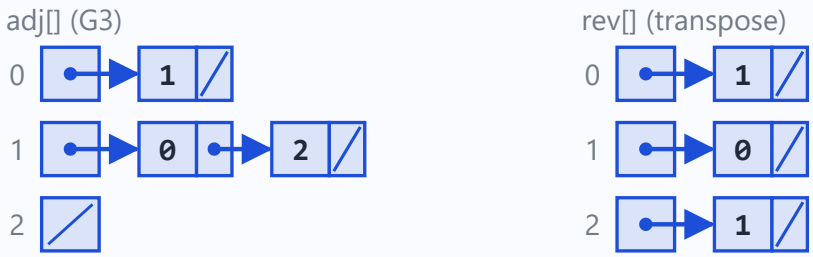


Figure 15.12: The adjacency list of G3 (left) and its inverse built with `transpose` (right). The 1 in `rev[2]` says that 2 is reached from 1.

The outer loop runs n times; the inner loops look at each item once in total. Each `insert` call is $O(1)$. The inverse list is therefore built in $O(n + e)$. Once built, the in-degree is also found in $O(d)$ by counting the items in `rev[i]`. The price is memory: every edge is stored twice, once in the out-list and once in the in-list.

15.3.5 Sequential representation in an array

Each item of an adjacency list carries a pointer. On a 64-bit machine this pointer takes twice the space of the 4-byte neighbor number. If the graph will never change, it is possible to get rid of the pointers altogether: all the lists are written one after another into a single array.

Definition 15.9

The **sequential representation** of an undirected graph with n vertices and e edges is a single array `int vertex[]` of $n + 2e + 1$ entries. The first $n + 1$ cells hold starting indices. The neighbors of i are in the cells `vertex[vertex[i]]`, ..., `vertex[vertex[i+1] - 1]`.

The array has two parts. The first $n + 1$ cells act like a table of contents: `vertex[i]` says where the neighbors of i begin in the array. The last $2e$ cells are the neighbor numbers themselves; the lists of the vertices are written in order, with no gaps between them. The neighbors of i end where the list of

the next vertex begins. So that the last vertex also has an end boundary, one extra cell, `vertex[n]`, is added. That is why the number of cells is *n + 1 + 2e*.



Figure 15.13: The sequential representation of G4. The cells with indices 0–8 are starting indices; `vertex[8] = 23` is the end boundary of the last vertex. The 14 cells with indices 9–22 hold the neighbor numbers.

For G4, *n = 8* and *e = 7*; the array has *8 + 14 + 1 = 23* cells. The neighbor entries start at index 9. The two neighbors of 0 are written into cells 9 and 10, the two neighbors of 1 into cells 11 and 12, and so on. Table 15.10 shows how to read the array, with examples.

Vertex *i*	`vertex[i]`	`vertex[i+1]`	Indices read	Neighbors	Degree
0	9	11	9, 10	1, 2	2
1	11	13	11, 12	0, 3	2
4	17	18	17	5	1
5	18	20	18, 19	4, 6	2
7	22	23	22	6	1

Table 15.10: Reading neighbors from the sequential representation of G4. The start is included and the end is excluded.

The routine that prints the neighbors of a vertex walks between the two bounds:

```
void print_adj(const int vertex[], int i)
{
    int k;
    for (k = vertex[i]; k < vertex[i + 1]; k++)
        printf("%d ", vertex[k]);
    printf("\n");
}
```

The array is taken as `const int vertex[]`; the routine only reads it, and the compiler prevents accidental writes. The loop variable `k` starts at `vertex[i]` and stops before reaching `vertex[i + 1]`. Table 15.11 traces the call `print_adj(vertex, 5)`.

Line	`k`	Condition `k < vertex[6]`	`vertex[k]`	Output
4–5	18	18 < 20 true	4	`4`
4–5	19	19 < 20 true	6	`4 6`

Line	k	Condition `k < vertex[6]`	`vertex[k]`	Output
4, 6	20	20 < 20 false	—	`4 6` and a newline

Table 15.11: Trace of `print_adj(vertex, 5)` (G4).

The loop runs as many times as the degree of the vertex: *O(d)*. The degree itself is found without even a loop, as the difference `vertex[i+1] - vertex[i]`, in *O(1)*. Memory is *n + 2e + 1* integers, with no pointers at all. For G4, 23 integers, that is 92 bytes, are enough. The adjacency list of the same graph needs 8 heads (64 bytes) and 14 items (224 bytes).

The array can be built from the existing lists in a single pass:

```
int build_sorted(int vertex[], int n)   /* one array from the lists */
{
    int i, k = n + 1;                  /* neighbors start at n+1 */
    node_ptr p;
    for (i = 0; i < n; i++) {
        vertex[i] = k;                  /* where i's first neighbor is */
        for (p = graph[i]; p; p = p->link)
            vertex[k++] = p->vertex;
    }
    vertex[n] = k;                      /* end bound of the last vertex */
    return k;                          /* cells used: n + 2e + 1 */
}
```

For each vertex the routine first writes the starting index, then copies the neighbors in its list into the next free cells. When the loop ends, `k` equals the number of cells used, that is *n + 2e + 1*. Writing this value into `vertex[n]` sets the boundary of the last vertex. Applied to the sorted lists of G4, it produces exactly the array of Figure 15.13.

The price of this representation is change. Adding a new edge means opening one cell in each of two lists. This requires shifting all the cells behind and updating the starting indices of the shifted vertices: *O(n + e)*. For a graph with many insertions and deletions this structure is not efficient. Storage is one matter; the efficiency of the algorithms that will work on the stored data is another. For example, the computer network in a building hardly ever changes. If the problem is only to hold this network and run queries on it, the sequential representation takes little space and is also read quickly because it is contiguous in memory. The compressed sparse row format (CSR) for sparse matrices is the same idea applied to a matrix.

15.3.6 Orthogonal representation of directed graphs

If both out-degree and in-degree are asked for often in a directed graph, the adjacency list and its inverse must be kept together. Then each edge is stored with two items, *2e* items in total. The orthogonal representation keeps each edge in a single item and links this item into two lists at the same time.

Definition 15.10

In the **orthogonal list** representation there is a single item for each edge *<tail, head>* of the directed graph. The item carries four fields: the tail, the head, a *row link* (the next edge leaving the same tail) and a *column link* (the next edge entering the same head). Each vertex has a row header (its out-list) and a column header (its in-list).

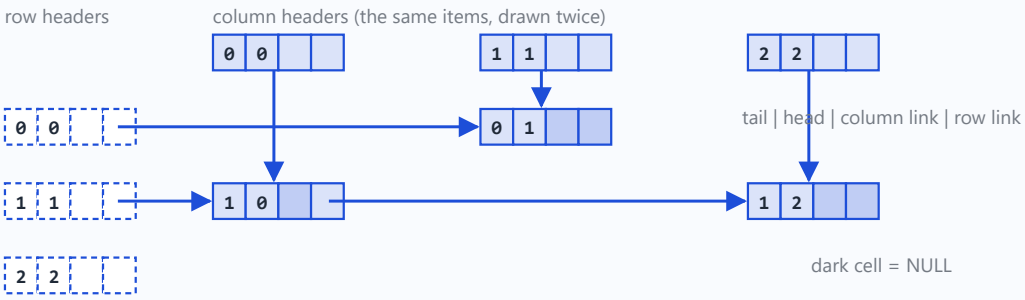


Figure 15.14: The orthogonal representation of G3. The row headers are on the left and the column headers on top. Each edge item is in one row (out) list and one column (in) list. The dark cells are `NULL` links.

The name of the representation comes from the look of the structure: the items are arranged in rows and columns like the filled cells of a matrix. Walking along the row of a vertex gives *where one can go* from that vertex, and walking along its column gives *where one comes from* into it. One item per edge is enough. In G3 the item *<1,0>* is in the row of 1 and in the column of 0.

This structure is very similar to the linked representation of sparse matrices. In a sparse matrix, too, each nonzero value is one item and is linked into its own row and column lists. As needed, the structure can be built with circular or doubly linked lists or with header nodes. Out-lists and in-lists are walked in $O(d)$; insertion at the front is $O(1)$.

The origin of this representation shows an instructive way of thinking. The solution that keeps the list together with its inverse is correct, but it uses $2e$ items. This waste is irritating and raises the question “can I do the same job in less space?” The first solution that comes to mind, and its first code, should never be trusted completely. Could the problem have been solved more efficiently if the underlying data structure were something else? Asking this question is the first step of good data structure design.

15.3.7 The multilist

In the adjacency list of an undirected graph, too, each edge is stored with two items. If some information is to be attached to the edges (for example, a “this edge has been processed” mark), the two copies of this information must be kept consistent. The multilist removes this duplication.

Definition 15.11

In the **adjacency multilist** representation there is a single item for each edge (v_i, v_j) of an undirected graph. The item sits in the edge list of both v_i and v_j. The basic item is the edge, not the vertex.

```
typedef struct edge *edge_ptr;
typedef struct edge {
    short marked;        /* processed? */
    int vertex1, vertex2;  /* endpoints */
    edge_ptr list1;    /* next for vertex1 */
    edge_ptr list2;    /* next for vertex2 */
} edge;
edge_ptr graph[MAX_V];
```

- **Line 3.** `marked` records whether the edge has been processed. Because the edge appears in two lists, a traversal meets the same edge twice; the mark prevents the edge from being handled again at the second encounter.
- **Line 4.** `vertex1` and `vertex2` are the two endpoints of the edge.
- **Lines 5–6.** `list1` points to the next edge in the list of `vertex1`; `list2` points to the next edge in the list of `vertex2`. An item is in the chains of two separate lists at the same time.
- **Line 8.** `graph[v]` points to the first item in the edge list of v. The head of a vertex with no edges is `NULL`.

If data must be attached to vertices (for example a city name), a separate vertex array is added to the structure, or a field such as `char label` to the item. On x86-64 this `edge` structure is 32 bytes with padding. In an adjacency list the same edge takes $2 \cdot 16 = 32$ bytes with two items. The gain is not in memory but in the information living in a single place.

Table 15.12 gives the multilist of G1. The six edges are the items N0–N5.

Item	vertex1	vertex2	list1	list2	In which lists
N0	0	1	N1	N3	0 and 1
N1	0	2	N2	N3	0 and 2
N2	0	3	NULL	N4	0 and 3
N3	1	2	N4	N5	1 and 2
N4	1	3	NULL	N5	1 and 3
N5	2	3	NULL	NULL	2 and 3

Table 15.12: The multilist of G1. Heads: `graph[0]` = N0, `graph[1]` = N0, `graph[2]` = N1, `graph[3]` = N2.

To read the table, follow the list of a vertex. The list of 1 starts at `graph[1]` = N0. In N0, 1 is `vertex2`; the next edge is `list2` = N3. In N3, 1 is `vertex1`; the next edge is `list1` = N4. In N4, 1 is again `vertex1`, and the list ends with `list1` = NULL. The edges of 1 are N0, N3, N4; its neighbors are 0, 2 and 3.

Traversing a multilist

The traversal rule is short: if *v* is in the item's `vertex1` field, follow `list1`; if in `vertex2`, follow `list2`. When `NULL` is seen, the list ends.

```
void print_adj(int v)
{
    edge_ptr p = graph[v];
    while (p) {
        if (p->vertex1 == v) {       /* v: vertex1 */
            printf("%d ", p->vertex2);  p = p->list1;
        } else {                     /* v: vertex2 */
            printf("%d ", p->vertex1);  p = p->list2;
        }
    }
}
```

Two things are done at each item: the other endpoint is printed and the right link is chosen. Whichever endpoint *v* is, the other endpoint is the neighbor, and the link to follow is the one on *v*'s side. Table 15.13 traces the call `print_adj(2)`.

Line	`p`	Item	which endpoint is 2	Printed	Next `p`
3	N1	(0,2)	—	—	—
5, 8	N1	(0,2)	`vertex2`	0	`list2` = N3
5, 8	N3	(1,2)	`vertex2`	1	`list2` = N5
5–6	N5	(2,3)	`vertex1`	3	`list1` = NULL
4	NULL	—	—	loop ends; output `0 1 3`	

Table 15.13: Trace of `print_adj(2)` on the multilist (G1).

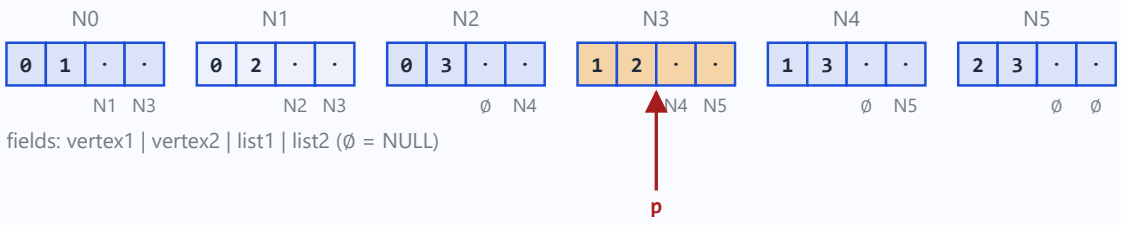


Figure 15.15: The multilist of G1 in memory. Each box has four fields: `vertex1`, `vertex2`, `list1`, `list2`; the names beneath them are the items the links point to. In the trace, `p` has reached N3, the second edge of 2.

The loop runs once for each edge of v: $O(d_v)$. Each edge is stored in memory only once, but it can be reached from the lists of both its endpoints.

Adding an edge to a multilist

A new edge enters two lists at once. Appending to the ends of the lists is possible, but it requires walking to `NULL` in both lists. Inserting at the front removes this cost:

```
void ml_add_edge(int u, int v)
{
    edge_ptr p = malloc(sizeof(edge));
    if (p == NULL) exit(1);
    p->marked = 0;
    p->vertex1 = u;  p->vertex2 = v;
    p->list1 = graph[u];  /* into u's list */
    p->list2 = graph[v];  /* into v's list */
    graph[u] = graph[v] = p;  /* at the front of both */
}
```

- **Lines 3–4.** Memory is allocated for the item and checked.
- **Lines 5–6.** The edge has not been processed yet; the endpoints are written.
- **Line 7.** `list1` points to the old first edge of u. It does not matter which endpoint of that old edge u is; the traversal routine determines this itself at every item.
- **Line 8.** `list2` points to the old first edge of v.
- **Line 9.** Both heads point to the new item. Lines 7 and 8 must be executed before this assignment; otherwise the old lists are lost.

Insertion is $O(1)$. For an undirected graph a single array of heads is enough. For a directed graph one keeps, as in the orthogonal representation, a head array together with a tail array. If the edges of G1 are added with `ml_add_edge` in the order (0,1), (0,2), ..., (2,3), the lists come out in the reverse of the arrival order, and this time `print_adj(2)` prints `3 1 0`. Table 15.12, on the other hand, shows a layout in which edges are appended at the ends of the lists; in that layout the output is `0 1 3`. Both structures are correct, but which code produces which drawing must be consistent. If the edges arrive in this order and each must be placed in $O(1)$, they are inserted at the head of the list. Drawing the list as if it were appended at the end would then contradict what the code does.

15.3.8 Application: four representations of the same graph

Example 15.2

For the directed graph with four vertices and edges *<0,1>, <0,3>, <1,2>, <2,0>, <3,2>*, build the adjacency matrix, the adjacency list, the inverse list and the orthogonal (single-item) list.

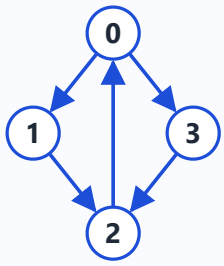


Figure 15.16: The directed graph of the application. 0 → 1 → 2 → 0 is a directed cycle.

Matrix. Write *a[i][j] = 1* if *<i, j>* exists; the row shows the tail and the column the head (Table 15.14). The row sums give the out-degrees (2, 1, 1, 1) and the column sums the in-degrees (1, 1, 2, 1). Each of the two sums is 5; in a directed graph every edge contributes to one out-degree and one in-degree.

	0	**1**	**2**	**3**	**out**
0	0	1	0	1	2
1	0	0	1	0	1
2	1	0	0	0	1
3	0	0	1	0	1
in	1	1	2	1	5

Table 15.14: The adjacency matrix of the application graph.

List and its inverse. Out-lists: 0 → 1, 3; 1 → 2; 2 → 0; 3 → 2. In-lists (transpose): 0 ← 2; 1 ← 0; 2 ← 1, 3; 3 ← 0. Each has *2e = 10* items in total.



Figure 15.17: The adjacency list of the application graph (outgoing edges, left) and its inverse (incoming edges, right).

Single-item representation. Each edge is a single item: N1 = <0,1>, N2 = <0,3>, N3 = <1,2>, N4 = <2,0>, N5 = <3,2>. Each item is in the out-list of its tail and the in-list of its head. The tail link points to the next edge leaving the same tail; the head link to the next edge entering the same head. Only *e = 5* items are needed.

Item	tail	head	tail link	head link		Vertex	out	in
N1	0	1	N2	NULL		0	N1	N4
N2	0	3	NULL	NULL		1	N3	N1
N3	1	2	NULL	N5		2	N4	N3
N4	2	0	NULL	NULL		3	N5	N2
N5	3	2	NULL	NULL				

Table 15.15: The single-item representation of the application graph. Edge items on the left, vertex headers on the right.

As a check, follow the in-list of 2: the header is N3 (<1,2>), the head link of N3 is N5 (<3,2>), and the head link of N5 is `NULL`. 2 is entered from 1 and from 3; the in-degree is 2, which agrees with column 2 of the matrix. The out-list of 0 starts at N1, moves to N2 through the tail link and ends: from 0 one can go to 1 and to 3.

15.3.9 Comparing the representations

Table 15.16 puts the five representations side by side under the same criteria. d_u is the degree of u.

Representation	Memory	Is (u, v) present?	Neighbors of u	Insert / delete
Adjacency matrix	$O(n^2)$	O(1)	O(n)	O(1)
Adjacency list	O(n + 2e)	$O(d_u)$	$O(d_u)$	O(1) at the front; O(d) search to delete
Sequential array	n + 2e + 1 integers	$O(d_u)$	$O(d_u)$	expensive: shifting O(n + e)
Orthogonal list (directed)	O(n + e)	O(d)	in and out O(d)	O(1) at the front
Multilist	O(n + e)	$O(d_u)$	$O(d_u)$	O(1) at the front

Table 15.16: Memory and operation costs of graph representations.

No row of the table is best in every column. The matrix is unrivaled for edge queries but slow at scanning neighbors and in memory. Lists are fast for traversal but slow for edge queries. The sequential array takes the least space but cannot cope with change. The choice depends on which operation will be performed how many times in the problem.

The lesson of this section is a method: first think about how the data will be stored, then write the code. The first solution that comes to mind may not be a good method. If the data structure is well designed, even an algorithm with n^3 steps can sometimes drop to n steps. The first example of this appears right below: the same traversal algorithm takes $O(n^2)$ with a matrix and $O(n + e)$ with a list. Storing data is not as easy a job as it looks.

15.3.10 Weighted graphs

In many problems it is not enough for an edge merely to exist; it also has a value. In a road network this value is distance, in a computer network latency, in a project plan duration, in cable laying cost. A graph whose edges have been assigned numbers is a **weighted graph**, and these numbers are called **weights**.

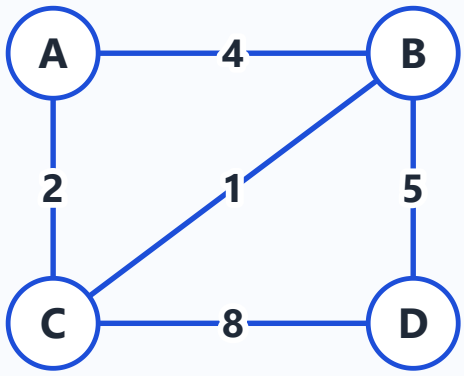


Figure 15.18: A weighted graph with four vertices. The numbers on the edges are the weights.

Both basic representations extend easily. In the matrix, the weight of the edge is written instead of 1; such a matrix is called a **cost matrix**. If there is no edge, a value meaning “infinity” is written into the cell. The diagonal is 0: going from a vertex to itself costs nothing. In the list, a `weight` field is added to each item.

	A	B	C	D
A	0	4	2	∞
B	4	0	1	5
C	2	1	0	8
D	∞	5	8	0

Table 15.17: The cost matrix of the graph in the figure. A pair with no edge is shown as ∞.

```c
#define INF 1000000   /* not INT_MAX: sums overflow */
int cost[MAX_V][MAX_V];

typedef struct wnode *wnode_ptr;
typedef struct wnode {
    int vertex;
    int weight;        /* cost of the edge */
    wnode_ptr link;
} wnode;
wnode_ptr wgraph[MAX_V];
```

C has no integer called infinity; a large constant is chosen. The first thing that comes to mind is `INT_MAX`, but this is a dangerous choice. Shortest path algorithms constantly compute sums of the form $d + w$. If any positive w is added while d = `INT_MAX`, the signed integer overflows. Signed overflow is undefined behavior in C; in practice the result is usually a large negative number, and the algorithm takes the vertex "at infinite distance" for the nearest one. That is why `INF` is chosen with care. It is larger than all real weights, but even twice its value fits in an `int`. $2 \cdot 1{,}000{,}000$ is far below the limit of a 32-bit `int` (about 2.1 billion).

The list item `wnode` is `node` with an added `int weight`; on x86-64 two `int`s take 8 bytes and the pointer 8 bytes, with no padding needed: 16 bytes. All minimum spanning tree and shortest path algorithms use one of these two representations.

15.4 Traversals

The most basic operation on a graph is traversal. Let an undirected graph $G = (V, E)$ and a vertex v be given. We want to visit every vertex reachable from v exactly once. "Visiting" a vertex means doing the work associated with it; in this chapter it means printing the vertex's number. Many algorithms, such as finding components, looking for cycles, shortest paths and spanning trees, are built on top of a traversal.

Traversal is easy in trees, because a tree has no cycles and every vertex is reached from the root by a single path. In a graph, a vertex can be reached by many paths. So the first rule of graph traversal is this: a visited vertex is marked, and a marked vertex is never entered again. Without the marks, the traversal would go around a cycle forever.

There are two basic traversal methods:

- **Depth-first search** (DFS). Go to a neighbor and from there dive as deep as possible. It resembles preorder traversal in a tree.
- **Breadth-first search** (BFS). First visit all neighbors of the start, then their neighbors. It resembles level-order traversal in a tree.

In both methods the graph is implemented with adjacency lists, and the lists are sorted. An example graph with eight vertices is used (Figure 15.19).

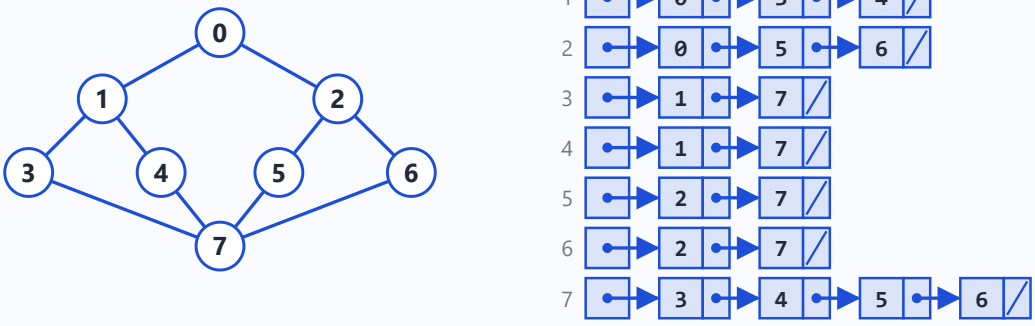


Figure 15.19: The example graph for the traversals (left) and its sorted adjacency list (right). The graph has 8 vertices and 10 edges; the lists contain *2e = 20* items.

15.4.1 Depth-first search

The idea of DFS resembles paying out a thread while moving through a maze. *v* is visited. Then an unvisited *w* is chosen from the list of *v*, and the search continues through *w*. If *w* also has an unvisited neighbor, the search moves on to it. When the search arrives at a vertex *u* with no unvisited neighbor left, it steps back and continues in the list of the vertex from which *u* was reached, from the place where it left off.

For these returns, one has to remember where each vertex's list was left off. Because the last one in is the first one out, this information is kept on a stack: while going deeper, vertices are pushed onto the stack, and when there is nowhere left to go, one is popped and the search continues. DFS written with an explicit stack looks complicated. The recursive routine does the same job very briefly, because the compiler's call stack holds the stack. The local variable `w` of each call stores where the list of that vertex was left off.

First, the common declarations:

```
#include <stdio.h>
#include <stdbool.h>
#define TRUE  true
#define FALSE false
bool visited[MAX_V];   /* don't forget ';' */
```

`stdbool.h` has provided the type `bool` and the values `true` / `false` since C99. The macros `TRUE` and `FALSE` are there to keep the notation of the textbook. `visited` is a global array; all its elements are `false` when the program starts. If the traversal is to be restarted, the array must be cleared by

hand.

```
void dfs(int v)   /* depth-first search */
{
    node_ptr w;
    visited[v] = TRUE;    /* visited */
    printf("%d ", v);
    for (w = graph[v]; w; w = w->link)
        if (!visited[w->vertex])
            dfs(w->vertex);
}
```

- **Line 3.** `w` is the pointer that moves along the list of *v*. Each recursive call has its own `w`.
- **Line 4.** *v* is marked. The mark is set before the vertex's list is scanned. That way, when *v* is seen in the list of one of its neighbors, it is not entered again.
- **Line 5.** Visit: the vertex number is printed.
- **Line 6.** The list of *v* is walked from start to end. The condition `w` in the `for` header means `w != NULL`.
- **Lines 7–8.** If the neighbor is not marked, the search continues through it. When the recursive call returns, the loop continues from the place that the `w` of this call points to, that is, from where it left off.

15.4.2 Tracing DFS

The call `dfs(0)` is traced on the example graph. In Table 15.18 the call stack is written from `dfs(0)` at the bottom up to the active call at the top. In the rows marked “skipped”, the neighbor is already marked.

Step	Event	Call stack	Output
1	`dfs(0)` : 0 is marked and printed	0	0
2	0 → 1 unmarked: `dfs(1)`	0 1	0 1
3	`w` → 0 in the list of 1: marked, skipped	0 1	0 1
4	1 → 3 unmarked: `dfs(3)`	0 1 3	0 1 3
5	`w` → 1 in the list of 3: skipped	0 1 3	0 1 3
6	3 → 7 unmarked: `dfs(7)`	0 1 3 7	0 1 3 7
7	`w` → 3 in the list of 7: skipped	0 1 3 7	0 1 3 7
8	7 → 4 unmarked: `dfs(4)`	0 1 3 7 4	0 1 3 7 4
9–10	list of 4: 1 and 7 are marked; `dfs(4)` ends	0 1 3 7	0 1 3 7 4
11	7 → 5 unmarked: `dfs(5)`	0 1 3 7 5	0 1 3 7 4 5
12	5 → 2 unmarked: `dfs(2)`	0 1 3 7 5 2	0 1 3 7 4 5 2
13–14	0 and 5 in the list of 2: skipped	0 1 3 7 5 2	0 1 3 7 4 5 2

Step	Event	Call stack	Output
15	2 → 6 unmarked: `dfs(6)`	0 1 3 7 5 2 6	0 1 3 7 4 5 2 6
16–17	list of 6: 2 and 7 are marked; `dfs(6)` and `dfs(2)` end	0 1 3 7 5	same
18	resuming in 5: `w` → 7 skipped; `dfs(5)` ends	0 1 3 7	same
19	resuming in 7: `w` → 6 skipped; `dfs(7)` and `dfs(3)` end	0 1	same
20	resuming in 1: `w` → 4 skipped; `dfs(1)` ends	0	same
21	resuming in 0: `w` → 2 skipped; `dfs(0)` ends	—	0 1 3 7 4 5 2 6

Table 15.18: Trace of `dfs(0)` on the example graph.

The most important moments of the trace are the returns. In step 12 the search moves from 5 to 2. There is a direct edge between 0 and 2. Still, 2 was reached by the long path 0 → 1 → 3 → 7 → 5 → 2. DFS tries depth first, not nearness. In step 18, `dfs(5)` resumes from the place where it was before descending to 2, with the second item in its list, 7. What makes this possible is the `w` kept in the frame of `dfs(5)`. The deepest moment of the stack is step 15: seven calls are open at the same time.

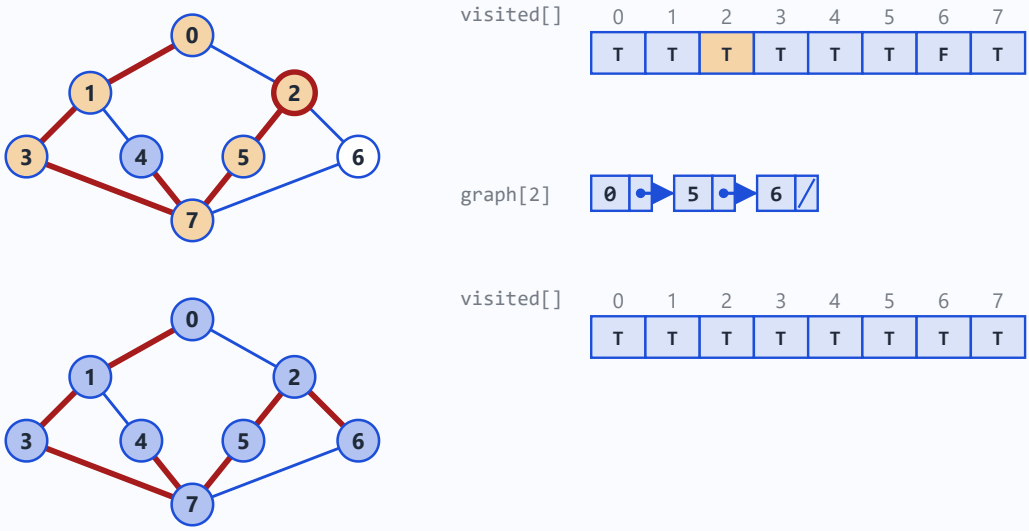


Figure 15.20: Two moments of DFS. Top: `dfs(2)` has just been called; the thick red edges are the tree edges followed so far, and on the right the `visited` array and the list of 2 are shown. Bottom: the traversal is finished; the tree edges form the depth-first spanning tree.

The edges through which a vertex is entered for the first time during the traversal form a tree: 0–1, 1–3, 3–7, 7–4, 7–5, 5–2, 2–6. The seven edges connect all eight vertices without a cycle. This tree is called the **depth-first spanning tree**. If one of the unused edges, for example 0–2, is added to the tree, a cycle forms and the structure stops being a tree.

15.4.3 What the result of DFS depends on

Besides the graph itself, the order that DFS produces depends on two things.

The start vertex. The start does not have to be 0. On the same graph with the same lists, `dfs(6)` prints `6 2 0 1 3 7 4 5` and `dfs(4)` prints `4 1 0 2 5 7 3 6`. The visited vertices are the same, but their order differs. In a connected graph every start reaches all the vertices.

The order of the lists. If the list of 0 were {2, 1} instead of {1, 2}, `dfs(0)` would descend to 2 first and the order would be `0 2 5 7 3 1 4 6`. Whether the list was built by inserting at the front or at the end, or kept sorted, directly changes the traversal. For this reason a DFS output can be stated exactly only when the exact order of the lists is known.

> **Theorem 15.2 (Cost of DFS)**
>
> On a graph with n vertices and e edges implemented with adjacency lists, the total running time of `dfs` is $O(n + e)$. If it is implemented with an adjacency matrix, it is $O(n^2)$.

Proof. `dfs(v)` is called only while `visited[v]` is false, and its first action is to make this value true. So at most one call is made per vertex; the total number of calls is at most n. Each call scans the list of its own vertex once; the list of v has d_v items. By Theorem 15.1, the total length of all the lists is $2e$. The total work is $O(n + 2e) = O(n + e)$. In a matrix, finding the neighbors of a vertex requires reading the whole row, that is n cells; each of the n calls takes $O(n)$, and the total is $O(n^2)$. ■

On the example graph, an instrumented `dfs` made 8 calls and looked at exactly 20 items in the lists; these values agree with $n = 8$ and $2e = 20$. The number of edges can be at most $n(n - 1)/2$. In a complete graph e reaches the order of n^2, and the costs of the two representations become equal. In sparse graphs, however, the list does the traversal in linear time.

The extra memory must also be counted. The `visited` array has n entries. The depth of the call stack is n in the worst case: in a graph whose vertices are laid out like a chain, each call opens the next. In very large graphs this depth can overflow the call stack; in that case DFS is written with an explicit stack.

15.4.4 Breadth-first search

BFS spreads like the rings made by a stone thrown into water. It starts from v and marks v. Then all the vertices in the list of v are visited. Next the unvisited neighbors of the first vertex in the list of v are visited, then those of the second, and so on. Every visited vertex is put into a queue. When the list of a vertex is finished, the vertex at the front of the queue is taken and its list is processed. When the queue becomes empty, the search ends.

By the nature of the method a queue is required, just as a stack is required in DFS. The neighbors of the vertex visited first must be visited first; “first in, first out” is exactly this rule.

```
void bfs(int v)   /* breadth-first search */
{
    node_ptr w;
    queue_ptr front = NULL, rear = NULL;
    printf("%d ", v);
    visited[v] = TRUE;
    addq(&front, &rear, v);
    while (front) {
        v = deleteq(&front, &rear);
        for (w = graph[v]; w; w = w->link)
            if (!visited[w->vertex]) {
                printf("%d ", w->vertex);
                addq(&front, &rear, w->vertex);
                visited[w->vertex] = TRUE;
            }
    }
}
```

- **Line 4.** The queue is implemented with a linked list; `front` and `rear` are initially `NULL`, that is, the queue is empty.
- **Lines 5–7.** The start vertex is printed, marked and put into the queue.
- **Line 8.** The loop runs while the queue is not empty. In an empty queue `front` is `NULL`.
- **Line 9.** The vertex at the front of the queue is taken. The parameter `v` is reused here; its value at the start of the routine is no longer needed.
- **Lines 10–15.** The list of `v` is scanned. Every unmarked neighbor is printed, appended to the rear of the queue and marked.

The timing of the mark is important: a vertex is marked when it is added to the queue, not when it leaves the queue. That way, even if a vertex is seen again in the list of another vertex while it waits in the queue, it does not enter the queue a second time. The value of this detail becomes clear when one looks at the result of the faulty form.

The dynamic queue

The queue that BFS uses is a dynamic queue implemented with a linked list. Items are taken from the front and added at the rear; two pointers are kept for the two ends.

```
typedef struct queue *queue_ptr;
typedef struct queue {
    int vertex;
    queue_ptr link;
} queue;
void addq(queue_ptr *front,
          queue_ptr *rear, int d)
{
    queue_ptr p = malloc(sizeof(queue));
    if (p == NULL) exit(1);
    p->vertex = d;  p->link = NULL;
    if (*front) (*rear)->link = p;  else *front = p;
    *rear = p;
}
```

`addq` takes the addresses of both pointers so that it can change them (`queue_ptr *front`). This is the only way in C to change a pointer from inside a function. The link of the new item is `NULL`, because it goes to the end of the queue. If the queue is not empty, the link of the old last item points to the new item; if it is empty, the new item is also the front item. In both cases the rear points to the new item. The operation is *O(1)*.

```
int deleteq(queue_ptr *front,
            queue_ptr *rear)
{
    queue_ptr p = *front;   /* front != NULL */
    int d = p->vertex;
    *front = p->link;
    if (*front == NULL) *rear = NULL;
    free(p);
    return d;
}
```

`deleteq` expects a nonempty queue; the comment states this precondition, and the BFS routine guarantees it with `while (front)`. The value of the front item is saved, the front pointer moves to the next one and the item is freed. When the queue becomes empty, the rear must also be set to `NULL`. If it is not, `rear` keeps pointing at the item that was just freed. The next `addq` sees `*front` as empty and behaves correctly, but in the meantime `rear` is a dangling pointer. That is why a prototype taking only the front pointer, such as `int deleteq(queue_ptr *)`, is not enough; the address of `rear` is passed as well. The operation is *O(1)*.

15.4.5 Tracing BFS

Table 15.19 traces the call `bfs(0)` on the example graph. In the queue column the left end is the front and the right end the rear.

Step	v	Event	Queue (front ... rear)	Output
1	0	0 is printed, marked and enqueued	0	0
2	0	0 is taken; its list is {1, 2}	—	0
3–4	0	1 and 2 unmarked: printed, added, marked	1 2	0 1 2
5	1	1 is taken; its list is {0, 3, 4}; 0 is marked	2	0 1 2
6–7	1	3 and 4 are added	2 3 4	0 1 2 3 4
8	2	2 is taken; its list is {0, 5, 6}; 0 is marked	3 4	0 1 2 3 4
9–10	2	5 and 6 are added	3 4 5 6	0 1 2 3 4 5 6
11	3	3 is taken; its list is {1, 7}; 1 is marked	4 5 6	0 1 2 3 4 5 6
12	3	7 is added	4 5 6 7	0 1 2 3 4 5 6 7
13	4	4 is taken; its list is {1, 7}: both are marked	5 6 7	same
14	5	5 is taken; its list is {2, 7}: marked	6 7	same
15	6	6 is taken; its list is {2, 7}: marked	7	same
16	7	7 is taken; its list is {3, 4, 5, 6}: marked	—	same
17	—	queue empty: the search ends	—	0 1 2 3 4 5 6 7

Table 15.19: Trace of `bfs(0)` on the example graph.

In step 12, 7 enters the queue and is marked. In step 13, 7 is seen again in the list of 4, but because it is marked it is not added a second time. The same happens in the lists of 5 and 6. 7 can be reached from four different vertices, but it enters the queue once.

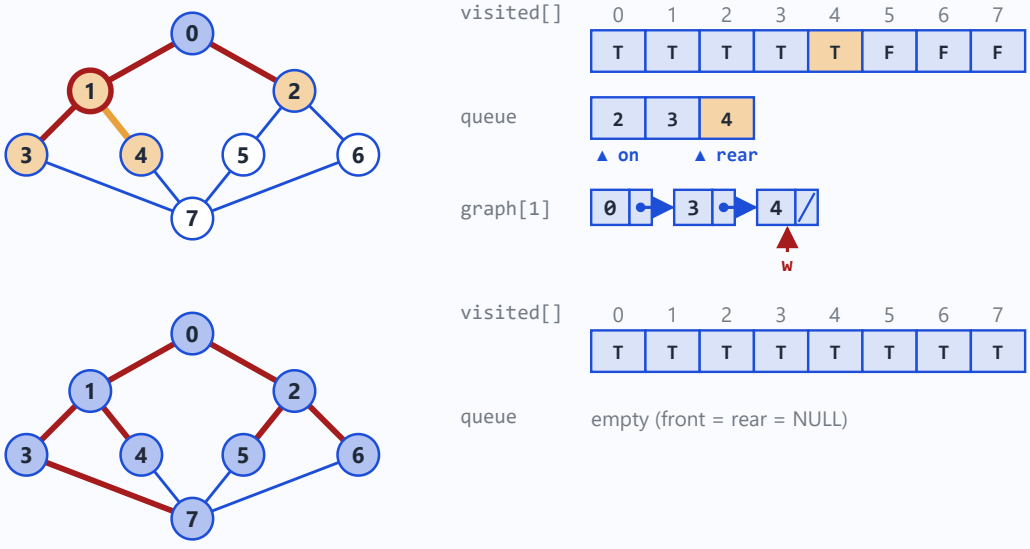


Figure 15.21: Two moments of BFS. Top: the list of 1 is being scanned; 3 and 4 have been added to the queue. Bottom: the search is finished; the red edges are the breadth-first spanning tree.

The output is `0 1 2 3 4 5 6 7`: the vertices are visited level by level, according to their number of edges from the start. 0 is at level 0; 1 and 2 at level 1; 3, 4, 5, 6 at level 2; 7 at level 3. The tree edges are 0–1, 0–2, 1–3, 1–4, 2–5, 2–6 and 3–7. In this **breadth-first spanning tree** the path from the root to each vertex is a path with the fewest edges in the graph.

Common mistake

Marking a vertex in BFS when it is taken out of the queue instead of when it is added. On the example graph, delete the line `visited[w->vertex] = TRUE`. In its place, write `visited[v] = TRUE` after `v = deleteq(…)`. The output is then `0 1 2 3 4 5 6 7 7 7 7`. Before it leaves the queue, 7 is seen once in each of the lists of 3, 4, 5 and 6, and it enters the queue four times. 11 insertions are made into the queue instead of 8, and 32 items are scanned in the lists instead of 20. Each duplicate scans its own list again when it leaves the queue. In denser graphs the work clearly exceeds the $O(n + e)$ bound.

15.4.6 Analysis of BFS and comparison with DFS

Theorem 15.3 (Cost of BFS)

On a graph implemented with adjacency lists, `bfs` runs in $O(n + e)$ time; with an adjacency matrix, in $O(n^2)$.

Proof. A vertex is added to the queue only while it is unmarked and is marked at the moment it is added. So each vertex enters the queue at most once, and the `while` loop runs at most n times. In each round the list of the vertex taken from the queue is scanned once. The total number of items scanned over all rounds is $d_0 + \dots + d_{n-1} = 2e$. `addq` and `deleteq` are $O(1)$. The total is $O(n + e)$. In a matrix each round reads the n cells of a row; over n rounds that comes to $O(n^2)$. ■

The extra memory of BFS is the `visited` array and the queue; at most n vertices are in the queue at once. The order that BFS produces is the order of the number of edges from the start. In an unweighted graph this means the order of the shortest paths; that is why BFS is the algorithm that finds paths with the fewest edges.

Depth-first and breadth-first search are two general search strategies that appear outside data structures too. In artificial intelligence, all the states of a problem form a search space, and this space is traversed as if it were a tree. Minimax search, B* search and branch and bound are built on these traversals. In a game problem, if one says “the possible moves are these” and expands every move one by one, the search is proceeding breadth-first.

Property	DFS	BFS
Auxiliary structure	Stack (the call stack in recursion)	Queue

Property	DFS	BFS
Progress	As deep as possible along one path	Layer by layer, from near to far
Counterpart in a tree	Preorder traversal	Level-order traversal
`(0)` on the example graph	0 1 3 7 4 5 2 6	0 1 2 3 4 5 6 7
`(7)` on the example graph	7 3 1 0 2 5 6 4	7 3 4 5 6 1 2 0
Cost (list / matrix)	$O(n + e)$ / $O(n^2)$	$O(n + e)$ / $O(n^2)$
Typical use	Components, cycle detection, topological sorting	Fewest-edge path, bipartiteness test

Table 15.20: Comparison of DFS and BFS.

15.5 Components, cycles and bipartiteness

The traversal routines alone only print vertex numbers. Their real value lies in the algorithms built on top of them. This section uses DFS and BFS to answer three questions: How many pieces does the graph consist of? Is there a cycle in the graph? Can the vertices of the graph be split into two groups so that every edge goes between the groups?

15.5.1 Connected components

Definition 15.12

A **connected component** of an undirected graph is one of the maximal connected subgraphs of the graph. “Maximal” means that if any other vertex of the graph is added to the component, the subgraph is no longer connected.

Every vertex belongs to exactly one component; the components partition the vertex set into disjoint parts. G4 has two components and G5 has nine. A connected graph has a single component, namely itself.

To find components, we use a property of traversal: `dfs(v)` marks all the vertices reachable from v, that is, the whole component of v. If an unmarked vertex remains when the traversal ends, it is in another component, and a new traversal is started from there.

```c
void connected(int n)
{
    int i;
    for (i = 0; i < n; i++)
        if (!visited[i]) {   /* new */
            dfs(i);
            printf("\n");
        }
}
```

- **Line 4.** All vertices are tried in turn.
- **Line 5.** An unmarked vertex is the first vertex of a new component that no earlier traversal could reach.
- **Line 6.** `dfs(i)` prints and marks all the vertices of this component.
- **Line 7.** Each component is printed on a separate line. If a counter increment (`count++`) is added at this line, the routine also gives the number of components; if the number is 1, the graph is connected.

The routine can be written in the same way with BFS instead of DFS. For the routine to work, the `visited` array must be entirely `FALSE` at the start.

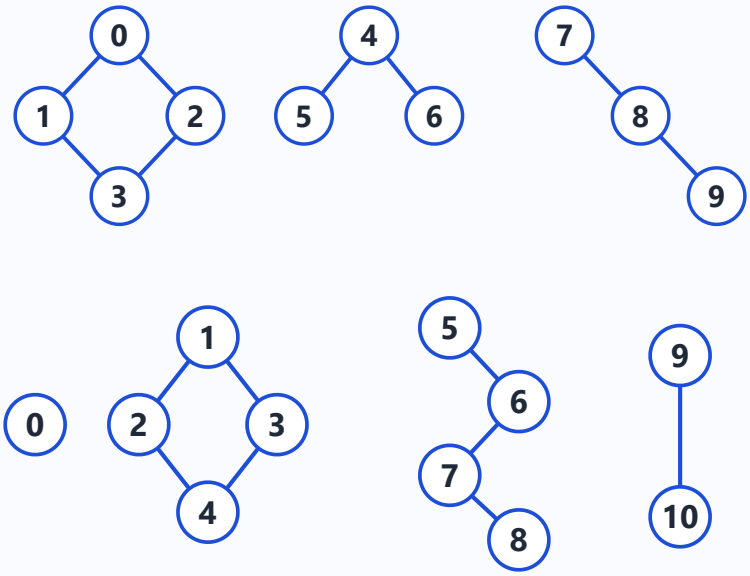


Figure 15.22: Two component examples. Top: a graph with 10 vertices and three components; the output is `0 1 3 2 / 4 5 6 / 7 8 9`. Bottom: a graph with 11 vertices and four components; the output is `0 / 1 2 4 3 / 5 6 7 8 / 9 10`.

In the bottom example the output of the second line is `1 2 4 3`. Since the list of 1 is {2, 3}, DFS goes first to 2, then to 4 in the list of 2, and from there to 3. 5 does not appear in this line; 5, 6, 7, 8 form another component. The component consisting of 0 alone has a single vertex; `dfs(0)` prints only 0,

because its list is empty. Table 15.21 is the trace of the top example.

`i`	`visited[i]`	Event	Line printed	Component
0	F	new component: `dfs(0)`	`0 1 3 2`	1
1, 2, 3	T	the `if` is not entered, O(1) each	—	1
4	F	new component: `dfs(4)`	`4 5 6`	2
5, 6	T	the `if` is not entered	—	2
7	F	new component: `dfs(7)`	`7 8 9`	3
8, 9	T	the `if` is not entered	—	3
10	—	loop ends	3 components	

Table 15.21: Trace of `connected(10)` (top graph).

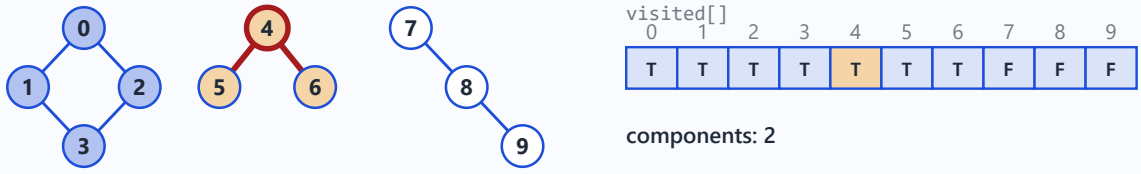


Figure 15.23: The moment of the trace at `i` = 4. The four vertices of the first component are marked (T); `dfs(4)` has traversed the second component.

15.5.2 Analysis: a sum, not a product

The cost of the routine is misleading at first sight. The outer `for` runs n times. The inner call `dfs` costs $O(n + e)$. Can these two numbers be multiplied and the complexity stated as $O(n(n + e))$? No. The rule "find the loops and multiply their costs" gives a wrong result here, because the cost of the inner work is not the same in every round.

Theorem 15.4 (Cost of finding components)

`connected` runs in $O(n + e)$ time with adjacency lists and in $O(n^2)$ time with an adjacency matrix.

Proof. All the `dfs` calls share the same `visited` array. Once a vertex is marked, no call enters it again. Hence all the `dfs` calls made during the `for` loop together visit each vertex once and scan each list once: $O(n + e)$ in total. Part of this total falls on the first component and part on the second; since the components are disjoint, no work is counted twice. To this are added the `for` loop's own n rounds; for marked vertices a round is $O(1)$. The total is $O(n) + O(n + e) = O(n + e)$. In a matrix every visit requires a row scan, so the total of the visits is $O(n^2)$. ■

The extreme case shows this clearly. If the graph is connected, in the first round `dfs(0)` traverses the whole graph in $O(n + e)$ and the whole `visited` array becomes `TRUE`. For the remaining $n - 1$ values of `i`, the `if` is never entered. The product would be right only if `dfs` traversed the whole graph again in every round; the marks prevent this. The same kind of reasoning applies to many algorithms in which loops are nested but the total work of the inner loop is bounded.

15.5.3 Connectivity and the number of edges

How many edges does a graph need in order to be connected?

Theorem 15.5 (Number of edges for connectivity)

A connected graph with n vertices has at least $n - 1$ edges. In a simple graph the number of edges is at most $n(n - 1)/2$. So in a connected simple graph

$$n - 1 \le e \le n(n - 1)/2$$

holds.

Proof. For the lower bound, imagine starting from a graph with no edges and adding edges one at a time. At first every vertex is a component by itself; there are n components. When an edge is added, its two endpoints are either in the same component, and the number of components does not change, or in different components, and the two components merge. An edge reduces the number of components by at most 1. To go from n components down to one, at least $n - 1$ edges are needed. For the upper bound we use the fact that in a simple graph there is at most one edge between each pair of vertices. Vertex 0 can be connected to the other $n - 1$ vertices. Vertex 1 can be connected to the $n - 2$ vertices after it. Continuing in this way gives $(n - 1) + (n - 2) + \dots + 1 = n(n - 1)/2$. ■

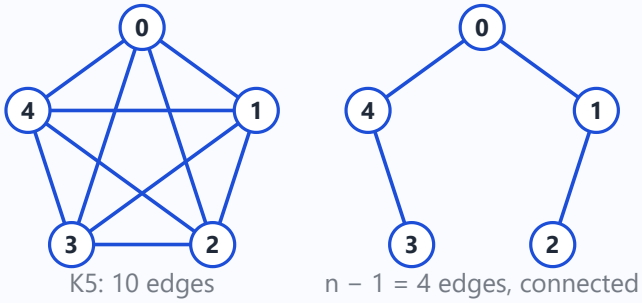


Figure 15.24: Two extreme cases, with 5 vertices. Left: the complete graph K5, with $5 \cdot 4 / 2 = 10$ edges. Right: a connected graph with 4 edges; $n - 1$ edges can be enough.

The theorem goes only one way. $n - 1$ edges are *necessary* for connectivity but *not sufficient*. In a graph with 5 vertices, 4 edges can form a chain; then the graph is connected. The same 4 edges can form a cycle among four vertices; then the fifth vertex is left alone and the graph is not connected. If the number of edges is less than $n - 1$, the graph is certainly not connected; in G5, $e = 2 < 10$.

The upper bound matters for complexity expressions. e can be a number of order n^2. So $O(n + e)$ equals $O(n^2)$ in a complete graph; a list beats a matrix asymptotically only in sparse graphs. A connected graph with exactly $n - 1$ edges contains no cycle; such a graph is a tree. The spanning trees produced by traversals also have $n - 1$ edges.

Game problems show the practical importance of these bounds. In a game such as tic-tac-toe, each state is a vertex and each move is an edge. The question "how do I counter my opponent's move?" amounts to a search in this graph. The number of states grows quickly, and the cost of the search depends directly on the numbers of vertices and edges.

15.5.4 Cycle detection in an undirected graph

A by-product of DFS is whether the graph has a cycle. While scanning the list of a vertex, DFS may meet a marked neighbor. In an undirected graph there can be two reasons. First, the neighbor was the vertex we just came from in the previous step; this is the other endpoint of the edge we arrived by and is not a sign of a cycle. Second, the neighbor was marked earlier through another path. In that case one can get from the vertex to that neighbor both along tree edges and along this edge: there is a cycle. Such edges are called **back edges**.

```
bool has_cycle(int v, int parent)
{
    node_ptr w;
    visited[v] = TRUE;
    for (w = graph[v]; w; w = w->link)
        if (!visited[w->vertex]) {
            if (has_cycle(w->vertex, v)) return TRUE;
        } else if (w->vertex != parent)
            return TRUE;          /* back edge */
    return FALSE;
}
```

- **Line 1.** Besides the vertex, the routine takes `parent`, that is, the vertex from which this vertex was reached. In the first call there is no parent; `-1` is used in this sense because it is not the number of any vertex.
- **Line 4.** The vertex is marked.
- **Lines 6–7.** The search descends into an unmarked neighbor; its parent is the current vertex. If the recursive call finds a cycle, `TRUE` is carried up immediately along the chain of calls.

- **Lines 8–9.** If the neighbor is marked and is not the parent, a back edge has been found. The `else if` belongs to the outer `if`, not to the inner one; the braces make this explicit. If the braces are removed, the `else` attaches to the nearest `if` and the meaning of the routine changes.
- **Line 10.** All neighbors have been passed and no cycle was found.

In the graph with four vertices and edges 0–1, 1–2, 2–3 and 3–1, the sorted lists are 0: {1}, 1: {0, 2, 3}, 2: {1, 3}, 3: {1, 2}. Table 15.22 traces the call `has_cycle(0, -1)`.

Call	`w`	State	Result
`has_cycle(0, -1)`	1	unmarked: enter	—
`has_cycle(1, 0)`	0	marked, but it is the parent	not counted as a cycle
`has_cycle(1, 0)`	2	unmarked: enter	—
`has_cycle(2, 1)`	1	marked, but it is the parent	not counted as a cycle
`has_cycle(2, 1)`	3	unmarked: enter	—
`has_cycle(3, 2)`	1	marked and not the parent (2)	back edge (3,1): `TRUE`

Table 15.22: Trace of `has_cycle(0, -1)`.

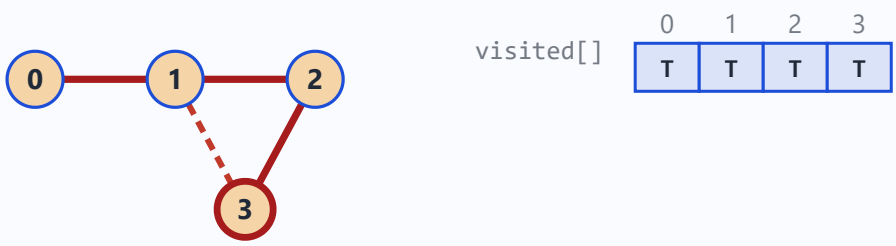


Figure 15.25: The last moment of cycle detection. The red edges are the DFS tree; the dashed edge (3,1) is the back edge and closes the cycle 1–2–3–1.

The value `TRUE` is carried from 3 to 2, from 2 to 1, from 1 to 0, and the routine ends early. If there is no cycle, the routine takes as long as DFS itself, $O(n + e)$. If the graph has several components, the routine is restarted from every unmarked vertex, as in finding components. On G2, which is known to be a tree, the routine returns `FALSE`.

15.5.5 Cycles in a directed graph

In a directed graph the parent check is not enough. An edge to a previously marked vertex need not close a cycle: that vertex may already have been finished in another branch. For example, with the edges 0 → 1, 0 → 2, 1 → 3, 2 → 3, DFS descends through 0, 1, 3, and then sees the edge from 2 to 3. 3 is marked, but there is no cycle. The distinguishing question is this: is the marked vertex on the stack right now? An edge to a vertex on the stack closes the path from that vertex down to the current vertex, and that is a cycle.

This distinction is made with three colors: **white** for a vertex not yet entered, **gray** for one that has been entered but not finished (it is on the stack), and **black** for one that is finished.

```
enum { WHITE, GRAY, BLACK };
int state[MAX_V];   /* all WHITE at the start */

bool directed_cycle(int v)
{
    node_ptr w;
    state[v] = GRAY;                  /* entered the stack */
    for (w = graph[v]; w; w = w->link) {
        if (state[w->vertex] == GRAY)
            return TRUE;              /* edge to a vertex on the stack */
        if (state[w->vertex] == WHITE && directed_cycle(w->vertex))
            return TRUE;
    }
    state[v] = BLACK;                 /* finished */
    return FALSE;
}
```

When a vertex is entered it is made gray. If a gray neighbor is seen, there is a cycle. A white neighbor is descended into. When all neighbors are done, the vertex becomes black and is no longer seen as part of any cycle. The routine returned `TRUE` on G3 ($0 \to 1 \to 0$) and on the application graph ($0 \to 1 \to 2 \to 0$), and `FALSE` on the four-vertex example above. The cost is again $O(n + e)$; with a matrix the same DFS is $O(n^2)$. The topological sorting routine also catches a cycle automatically.

The practical importance of cycle detection is great. In a project where precedence relations are defined between tasks, a directed cycle means a contradiction such as "A must finish before B, and B before A", and the project never ends. When building a minimum spanning tree, an edge that closes a cycle is a redundant edge and is not taken into the tree.

15.5.6 Testing for bipartiteness

Definition 15.13

A graph is **bipartite** if its vertex set V can be split into two disjoint sets A and B such that every edge has one endpoint in A and the other in B.

Students and courses form a bipartite graph: edges connect only a student to a course. $K_{3,3}$ is a bipartite graph in which each vertex of a group of three is connected to every vertex of the other group of three.

The test is a coloring problem. The start vertex is painted color 0. BFS paints each unmarked neighbor with the opposite color of its own. If the two endpoints of an edge turn out to have the same color, the graph is not bipartite. If no conflict arises, the colors give the two groups.

```
int color[MAX_V];   /* -1: not colored, 0 or 1 */

bool is_bipartite(int n)
{
    int i, v;
    node_ptr w;
    queue_ptr front = NULL, rear = NULL;
    for (i = 0; i < n; i++) color[i] = -1;
    for (i = 0; i < n; i++) {
        if (color[i] != -1) continue;      /* new component */
        color[i] = 0;
        addq(&front, &rear, i);
        while (front) {
            v = deleteq(&front, &rear);
            for (w = graph[v]; w; w = w->link)
                if (color[w->vertex] == -1) {
                    color[w->vertex] = 1 - color[v];
                    addq(&front, &rear, w->vertex);
                } else if (color[w->vertex] == color[v]) {
                    while (front) deleteq(&front, &rear);   /* empty the queue */
                    return FALSE;
                }
        }
    }
    return TRUE;
}
```

- **Line 1.** The `color` array holds both the mark and the color: `-1` is a vertex not yet colored. A separate `visited` array is not needed.
- **Lines 8–12.** All colors are cleared; BFS is started separately for each component; the color of one component does not affect another.
- **Lines 16–18.** An uncolored neighbor is painted the opposite color and enters the queue.
- **Lines 19–21.** If the neighbor is colored and has the same color, the test fails. Before leaving the routine the queue is emptied; otherwise the items left in the queue are never freed.

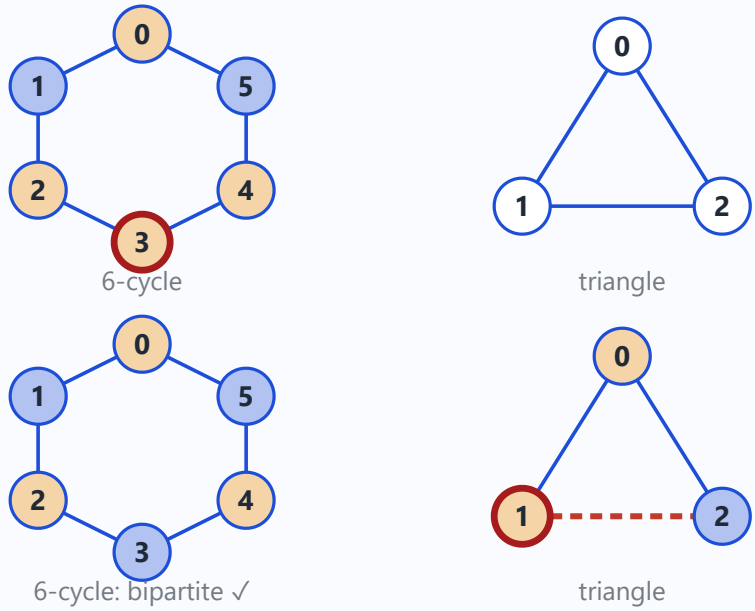


Figure 15.26: 2-colorings. Top: the 6-cycle colored by BFS: 0, 2, 4 have one color and 1, 3, 5 the other; the graph is bipartite. Bottom: the triangle: after 0 and 1 are painted different colors, 2 is adjacent to both, and two colors are not enough.

In the 6-cycle, BFS colors the vertices in the order 0, 1, 5, 2, 4, 3; the colors are 0, 1, 1, 0, 0, 1. No edge has both endpoints in the same color. In the triangle, 0 gets color 0, and its neighbors 1 and 2 get color 1. When the list of 1 is scanned, 2 is seen; both have color 1, and the test fails.

Theorem 15.6 (Bipartiteness)

A graph is bipartite if and only if it contains no cycle of odd length.

Proof. One direction is easy. If the graph is bipartite, every edge changes color. A cycle returns to the vertex it started from, that is, to the color it started with; so the number of color changes on the cycle, which is its number of edges, is even. There can be no odd cycle. The other direction follows from the BFS coloring producing a conflict. If two neighbors u and v of the same color are found, they have a common ancestor in the BFS tree. Since they have the same color, the difference of their distances from the start is even; since the levels of adjacent vertices in BFS can differ by at most 1, their levels are equal. The two paths descending from the common ancestor to u and to v have equal length; adding the edge (u, v) to them gives a cycle of odd length. So if there is no odd cycle, the coloring finishes without conflict. ■

The cost of the test is the cost of BFS: $O(n + e)$. The routine returned `TRUE` for the 6-cycle and for $K_{3,3}$, and `FALSE` for the triangle; the example traversal graph is bipartite too, because all of its cycles have 4 or 6 edges.

15.6 Complexity summary

Table 15.23 gives the costs of the operations studied in this chapter for the two basic representations. n is the number of vertices, e the number of edges and d the degree of the vertex in question. In a connected simple graph, $n - 1 \le e \le n(n-1)/2$.

Operation	Matrix	List	Reason
Memory	$O(n^2)$	$O(n + e)$	2e items in an undirected list
Is (u, v) present?	$O(1)$	$O(d_u)$	A single cell / a list scan
Degree (undirected)	$O(n)$	$O(d)$	Row / list length; for in-degree in a directed graph, the inverse list
Number of edges	$O(n^2)$	$O(n + e)$	Sum of degrees / 2
DFS, BFS	$O(n^2)$	$O(n + e)$	Each vertex once, each list once
Connected components	$O(n^2)$	$O(n + e)$	Sum of the traversals; not a product
Cycles, bipartiteness	$O(n^2)$	$O(n + e)$	Constant extra work on top of DFS or BFS
Building the transpose list	—	$O(n + e)$	The lists are walked once

Table 15.23: Costs of the basic graph operations with an adjacency matrix and an adjacency list.

The essence of the table is a single sentence: finding neighbors is $O(n)$ in a matrix and $O(d)$ in a list. All traversal-based algorithms add up this difference over n vertices; it comes to $n \cdot n$ in a matrix and $\Sigma d = 2e$ in a list.

15.7 Common mistakes

1. **Treating a directed graph as undirected.** Assuming $\langle v, u \rangle$ exists when $\langle u, v \rangle$ does, or assuming that the matrix of a directed graph is symmetric. In G3, $a[1][2] = 1$ and $a[2][1] = 0$.
2. **Setting the mark too late in BFS.** Marking a vertex when it is taken out of the queue instead of when it is added. The same vertex enters the queue several times; on the example graph 7 is printed four times.
3. **Not resetting the mark array.** `visited` is global. If it is not set back to `FALSE` before a second traversal, the second traversal cannot enter any vertex and prints only its start.
4. **Multiplying costs in the component analysis.** Multiplying the outer `for` by DFS and saying $O(n(n + e))$. Thanks to the marks, the total of all the DFS calls is $O(n + e)$.
5. **Careless memory management.** Not checking the result of `malloc`; allocating pointer-sized space with `sizeof(node_ptr)`; not calling `free` on the lists at the end of the program; reading an item's link after it has been freed.
6. **Needlessly inserting at the end.** Appending the edge to the end of the list and walking the list from the start every time. If order does not matter, the $O(1)$ insertion at the front is enough; the drawing must then show the reversed order that the code produces.

7. **Forgetting the size in a matrix parameter.** Writing `int a[][]` does not compile; the second dimension must be given: `int a[][MAX_V]`.
8. **Choosing `INT_MAX` for infinity.** Signed overflow occurs in cost sums; use a safely large constant.

15.8 Exercises and solutions

The exercises in this section are given in the order of the slides; each is followed by its solution.

15.8.1 Degree sequence

Question. Is there a simple undirected graph with degree sequence (3, 3, 2, 2, 1)? How many edges does it have?

Solution. The sum of the degrees is 11. By Theorem 15.1 the sum must be $2e$, that is, even. There is no such graph. The same conclusion follows from the number of odd-degree vertices: there are three, namely 3, 3 and 1, whereas this number must be even.

15.8.2 Neighbors in the sequential representation

Question. In the sequential representation of G4, which indices hold the neighbors of vertex 5?

Solution. `vertex[5] = 18` and `vertex[6] = 20`. The neighbors are in the cells with indices 18 and 19: `vertex[18] = 4`, `vertex[19] = 6`. The neighbors of 5 are 4 and 6, and its degree is $20 - 18 = 2$.

15.8.3 Traversals starting from 7

Question. On the eight-vertex example graph, what are the outputs if DFS and BFS are started from 7 (lists sorted)?

Solution. DFS: the list of 7 is {3, 4, 5, 6}; the search descends into the first neighbor, 3. The first unmarked neighbor of 3 is 1, of 1 it is 0, of 0 it is 2, and of 2 it is 5. The neighbors of 5, 2 and 7, are marked; the search returns to 2 and descends into 6. The returns continue up to 1, and 4 in the list of 1 is found. The output is `7 3 1 0 2 5 6 4`. BFS: the neighbors of 7, 3, 4, 5, 6, are printed in order and enqueued. When 3 is taken, 1 is added; when 4 is taken, nothing; when 5 is taken, 2 is added. The neighbors of 6 are marked. When 1 is taken, 0 is added. The output is `7 3 4 5 6 1 2 0`.

15.8.4 Moving the mark line

Question. What happens if, in the BFS code, the line `visited[w->vertex] = TRUE` is moved to after `deleteq`?

Solution. A vertex stays unmarked while it waits in the queue. When it is seen in the list of another vertex, it is printed again and added to the queue again. On the example graph, before leaving the queue 7 is added four times, via 3, 4, 5 and 6; `bfs(0)` prints `0 1 2 3 4 5 6 7 7 7 7`. If started from 7, 1, 2 and 0 are also repeated: `7 3 4 5 6 1 1 2 2 0 0 0 0`. The output is wrong and the work exceeds $O(n + e)$.

15.8.5 Counting components with a matrix

Question. Write a routine that finds the number of components of a graph given as an adjacency matrix; what is its complexity?

Solution. The matrix form of DFS finds the neighbors of a vertex by scanning its row:

```
void mdfs(int a[][MAX_V], int n, int v)
{
    int j;
    visited[v] = TRUE;
    printf("%d ", v);
    for (j = 0; j < n; j++)            /* the whole row: O(n) */
        if (a[v][j] && !visited[j])
            mdfs(a, n, j);
}

int matrix_components(int a[][MAX_V], int n)
{
    int i, count = 0;
    for (i = 0; i < n; i++) visited[i] = FALSE;
    for (i = 0; i < n; i++)
        if (!visited[i]) {
            count++;
            mdfs(a, n, i);
            printf("\n");
        }
    return count;
}
```

`mdfs` descends into each unmarked neighbor for every 1 in the row of v. `matrix_components` clears the marks, starts a new traversal from every unmarked vertex and increments the counter. Applied to the matrix of G4, it prints the lines `0 1 3 2` and `4 5 6 7` and returns 2. Each vertex is visited once, and each visit scans a row of n cells: $O(n^2)$.

15.8.6 Giving the in-degree in O(1)

Question. In a directed graph, which structure is kept to give the in-degree in $O(1)$, and how is it updated?

Solution. A counter `in[]` is kept for each vertex. When the edge $\langle u, v \rangle$ is added, `in[v]++` is done; when it is deleted, `in[v]--`. The query is a single array read. If the incoming edges themselves are also needed, an inverse list or an orthogonal list is kept; the length of the inverse list is the in-degree, but counting it is $O(d)$.

15.9 Summary

- A graph $G = (V, E)$ is a nonempty vertex set together with a possibly empty edge set. In an undirected graph $(u, v) = (v, u)$; in a directed graph $\langle u, v \rangle \neq \langle v, u \rangle$.
- Adjacency, subgraph, path, cycle, connectivity and degree are the basic notions. Degree is always stated for a particular vertex; in a directed graph it is the sum of in-degree and out-degree.
- The degrees sum to $2e$. In a connected graph $n - 1 \le e \le n(n-1)/2$; the lower bound is necessary but not sufficient.
- The adjacency matrix needs $O(n^2)$ memory; an edge query is $O(1)$ and a neighbor scan is $O(n)$. The adjacency list needs $O(n + e)$ memory; a neighbor scan is $O(d)$.
- Insertion at the front places each edge in $O(1)$ and reverses the list relative to the arrival order. In a directed graph, the inverse list for in-degrees is built in $O(n + e)$.
- The sequential array is a pointer-free representation of $n + 2e + 1$ integers but is unsuited to change. The orthogonal list and the multilist keep each edge in a single item.
- In a weighted graph the matrix gets the weights and the list gets a `weight` field; for infinity a large constant that does not overflow is chosen.
- DFS descends with recursion or a stack; BFS advances layer by layer with a queue. Both are $O(n + e)$ with lists and $O(n^2)$ with a matrix. The resulting order depends on the start vertex and the list order.
- Connected components are found in $O(n + e)$ by starting a traversal from every unmarked vertex; the costs of the outer loop and the traversal are not multiplied.
- In an undirected graph a marked neighbor that is not the parent, and in a directed graph an edge to a gray vertex, means a cycle. 2-coloring with BFS tests bipartiteness; a bipartite graph has no cycle of odd length.
- Method: first the problem and its operations, then how the data will be stored, and the code last. Do not trust the first representation that comes to mind.

15.10 Key terms

graph vertex edge directed graph adjacent subgraph path cycle connected degree in-degree out-degree complete graph adjacency matrix adjacency list orthogonal list adjacency multilist depth-first search breadth-first search spanning tree minimum spanning tree connected component cycle detection bipartite graph queue call stack shortest path

15.11 Test Yourself

1. What are the minimum and maximum numbers of edges in a connected simple graph with 9 vertices?

   Answer

   At least $n - 1 = 8$, at most $9 \cdot 8 / 2 = 36$. A connected graph with 8 edges is a tree; a graph with 36 edges is the complete graph K_9.

2. The adjacency lists of an undirected graph contain 30 items in total. How many edges does the graph have? If the same graph were stored as an adjacency matrix, how many 1s would the matrix contain?

   Answer

   Since every edge appears in two lists, $e = 30 / 2 = 15$. In the matrix, too, every edge is written into two symmetric cells: 30 ones.

3. The lists of G2 are kept sorted. What do `dfs(0)` and `bfs(0)` print?

   Answer

   DFS: `0 1 3 4 2 5 6`; this is the same as preorder traversal in the tree. BFS: `0 1 2 3 4 5 6`; the same as level-order traversal.

4. What does `bfs(6)` print on the eight-vertex example graph?

   Answer

   `6 2 7 0 5 3 4 1`. The neighbors of 6 are 2 and 7. When 2 is taken, 0 and 5 are added; when 7 is taken, 3 and 4. When 0 is taken, 1 is added; the neighbors of the rest are marked.

5. The eight-vertex example graph ($e = 10$) is stored in the sequential array representation. How many cells does the array have, and at which indices are the neighbors of 7?

   Answer

   $n + 2e + 1 = 8 + 20 + 1 = 29$ cells. The starting indices are 9, 11, 14, 17, 19, 21, 23, 25, 29. `vertex[7] = 25`, `vertex[8] = 29`: the neighbors of 7 are at indices 25–28: 3, 4, 5 and 6.

6. The edges of G1 are added to a multilist with `ml_add_edge` in the order (0,1), (0,2), (0,3), (1,2),

(1,3), (2,3). What does `print_adj(0)` print?

Answer

`3 2 1`. Every edge enters at the front of the lists; in the list of 0 the most recently added, (0,3), is at the front.

7. If `connected(11)` is applied to G5 (11 vertices; edges (1,4) and (7,8)), how many lines are printed? How many times is the `if` entered?

Answer

9 lines: `0`, `1 4`, `2`, `3`, `5`, `6`, `7 8`, `9`, `10`. The `if` is entered 9 times; at `i` = 4 and `i` = 8 the vertices were already marked in earlier traversals.

8. If only `else` is written instead of `else if (w->vertex != parent)` in `has_cycle`, what does the routine return for the tree G2? Why?

Answer

It returns `TRUE`, although G2 has no cycle. `has_cycle(1, 0)` sees 0, the first item in the list of 1, as marked and counts it as a back edge. In an undirected graph the edge we arrived by is also in the list of the other endpoint; the parent check is there to set this edge apart.

15.12 Further reading

- E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2nd ed., Chapter 6. The representations, the `dfs` and `bfs` routines and the connected components routine in this chapter follow its presentation; the details of the multilist and the sequential representation are also found there.
- T. H. Cormen et al., *Introduction to Algorithms*, 4th ed., Chapter 20. It gives the white/gray/black coloring of DFS, the classification of edges and the proofs of the shortest path property of BFS.
- R. Sedgewick, *Algorithms in C, Part 5: Graph Algorithms*, 3rd ed. Graph representations and traversals are treated with C code and with experiments on large graphs.

15.13 References

1. E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2nd ed., Silicon Press, 2008. Chapter 6, Graphs.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022. Chapter 20, Elementary Graph Algorithms.

3. R. Sedgewick, *Algorithms in C, Part 5: Graph Algorithms*, 3rd ed., Addison-Wesley, 2002.

PART IV · CHAPTER 16

Sets and Union-Find

Many problems reduce to the same two questions: do two items belong to the same group, and if not, how are the two groups merged into one? Network connectivity, counting components in a graph, equivalence declarations and building a minimum spanning tree ask these questions millions of times. The structure that keeps sets with no common elements and supports only these two operations is called union-find. This chapter represents the sets as trees linked from child to parent and stores the whole forest in a single integer array. It shows step by step which sequence of operations turns the first version into a chain and how the cost climbs to $O(n^2)$. Two improvements follow: the weighting rule, which bounds the tree depth by $\lfloor \lg m \rfloor + 1$, and the collapsing rule, which hooks the find path directly to the root. Together they bring the cost per operation down to a factor that, because of the inverse Ackermann function, never exceeds 4 in practice. For every routine the chapter gives the C code, a line-by-line walkthrough, a code trace, a measurement and a complexity analysis.

LEARNING OUTCOMES

Model
Represent pairwise disjoint sets as trees whose root is the set name and whose links run from child to parent.

Implement
Write the `find` and `union` routines with a single parent array; build the array with `malloc` and release it with `free`.

Find the worst case
Show which sequence of operations turns the tree into a chain and why the total cost is $O(n^2)$.

Improve
Apply the weighting rule and the collapsing rule; interpret the depth lemma together with its proof.

Analyze
Explain the Tarjan–van Leeuwen result and why the inverse Ackermann function means “almost constant”.

Use

Apply union-find to counting connected components, equivalence classes and Kruskal's algorithm.

16.1 Disjoint sets

Who is friends with whom in a social network, which intersections of a city are joined by road, which terminals of a circuit are soldered to the same conductor? In all these questions the items are divided into groups, and each item is in exactly one group. Over time the groups merge: when a road is built between two intersections, two regions become one. Groups are never seen to split. The structure needed for this kind of problem is very simple. It supports just two operations, but it performs them extremely fast. Written in a few lines, this structure shows up as a building block in many graph algorithms. It should be learned well first, so that it comes to mind and can be used when needed.

Before turning to programming, the problem must be stated correctly in the language of mathematics, as someone who does not know programming would state it. A correctly formed definition shows from the start which operations are needed and which are superfluous.

16.1.1 Disjoint sets and the forest representation

The items are named by the numbers $0, 1, \ldots, n - 1$. These numbers can be a student's ID, a computer's position in a network or the code of an intersection. The actual data of the items is kept in another array; union-find works only with the numbers.

Definition 16.1

Let the universe be $U = \{0, 1, \ldots, n - 1\}$. If the sets $S_1, S_2, \ldots, S_k$ are subsets of U and $S_i \cap S_j = \emptyset$ for every $i \neq j$, these sets are called **pairwise disjoint sets**. For short they are called **disjoint sets**. A data structure that supports the union and find operations on disjoint sets is called **union-find**.

The condition in the definition says that no item is in two sets at once. Every item has exactly one set; so the question “which set is the item in?” always has a single answer.

Example 16.1

For $n = 10$, the sets $S_1 = \{0, 6, 7, 8\}$, $S_2 = \{1, 4, 9\}$ and $S_3 = \{2, 3, 5\}$ are pairwise disjoint. The intersections $S_1 \cap S_2$, $S_2 \cap S_3$ and $S_1 \cap S_3$ are all empty. The union of the three sets gives the whole universe, that is, ten items. All the first examples in this chapter work with these three sets.

How should the sets be represented on a computer? The chosen way is to represent each set as a tree. One of the set's elements becomes the root and the others sit below it. There is one important difference from ordinary trees: the links point **from child to parent**. Each node knows only its own parent; it does not know its children. That is why the arrows in Figure 16.1 point upward. A collection of mutually disjoint trees is called a **forest**. The representation of disjoint sets is also a forest.

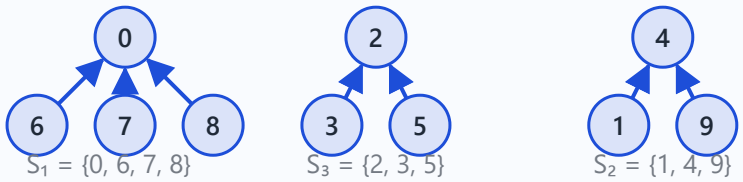


Figure 16.1: The three sets of Example 16.1 and their trees. Arrows go from child to parent. The roots (0, 2 and 4) have no parent.

The placement within the trees has no meaning. That 6, 7 and 8 sit below 0 does not mean that 0 is "special" in this set. The same set could be represented by a tree in which 7 is the root and the others are attached to it. The tree's only job is to carry the information "these items belong together". This freedom is very useful later: we can choose the shape of the tree as we like, in the direction that lowers the cost.

Why the links point upward also follows from the operations. Union-find never asks "list the elements of this set". The question is always "which set is this item in?" This question is answered by starting at an item and climbing to the root. To climb, it is enough for each node to know its parent. Keeping child lists would both waste memory and never be used by any operation.

16.1.2 Union and find

The structure supports two operations:

1. **Union of disjoint sets.** If S_i and S_j are two disjoint sets, their union is the set $S_i \cup S_j = \{x : x \in S_i \vee x \in S_j\}$. After the union, S_i and S_j no longer exist separately; a single set takes their place. For example, $S_1 \cup S_2 = \{0, 6, 7, 8, 1, 4, 9\}$.
2. **Find.** `find2(i)` finds the set that contains the element i. The root of the tree is returned as the name of the set.

Union is very easy with trees. One of the roots of the two trees is made a child of the other. A single link is enough for this: the parent information of one of the roots is updated to point to the other root. Figure 16.2 shows two forms of the same union. In the bottom left figure the parent of 4 has been made 0, and the tree of S_2 has become a subtree of 0. In the bottom right figure the parent of 0 has been made 4. Both results represent the same set, and both are correct. But the shapes of the trees differ. Which one is chosen determines the cost of later find operations. Most of this chapter is about how to make that choice.

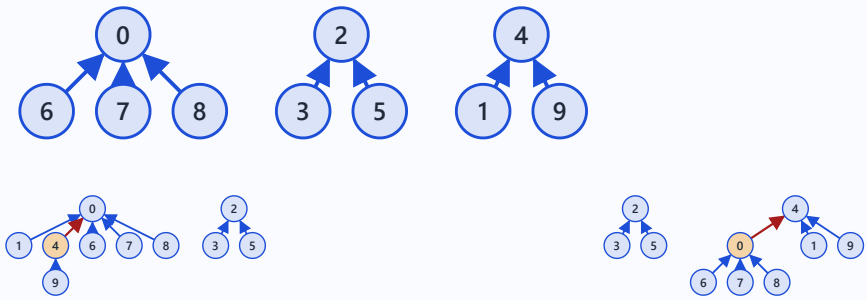


Figure 16.2: Top: the start, with roots 0, 2 and 4. Bottom left: $S_1 \cup S_2$, where the parent of 4 is made 0. Bottom right: $S_2 \cup S_1$, where this time the parent of 0 is made 4. The red arrow is the newly written link.

In the form on the right, the distance of items 6, 7 and 8 from the root has grown from 1 to 2. In the form on the left, only the distances of 1, 4 and 9 have grown, each by one. Finding the set of an item means walking from that item to the root. As the distance to the root grows, find becomes more expensive. The weighting rule rests on this small observation.

16.1.3 The name of the set: the root

Names like S_1, S_2 are for humans. Algorithms associate each set name with the root of the tree. Instead of "the set S_1" one says "set number 0". In this way the root number returned by `find2` becomes the name of the set.

Set name	Link (root)
S_1	$\rightarrow 0$
S_2	$\rightarrow 4$
S_3	$\rightarrow 2$

Table 16.1: Table of links from set names to roots. If names are needed as text, a separate table like this is kept.

Using the root as the name has an important consequence. Whether two items are in the same set is decided by comparing their roots. The question `find2(8) == find2(6)` means "are 8 and 6 in the same set?" Both calls return 0, so the answer is yes. `find2(8) == find2(9)` compares 0 with 4; the answer is no.

The name of a set can change with a union. After the union on the right of Figure 16.2, the set of 8 is no longer 0 but 4. So the set of an item should not be saved in a variable and used later. Every question is answered by calling `find2` again on the current trees.

Common mistake

Taking a root number to be the permanent identity of an item. Writing `int k = find2(8);` and using `k` after a few unions can give a wrong result. Since roots change places in a union, 0 may no longer be a root. A root is the name of the set only at the moment it was asked.

16.2 Array implementation

The model is ready: each set is a tree, the root of each tree is the name of the set, and the links go from child to parent. Now we must decide how to store this model in C.

16.2.1 Implementation options

The first way that comes to mind is to keep each set as a linked list. Joining two lists end to end is done by changing the pointer of the last node; union is *O(1)*. But to find out which list an item is in, the lists have to be scanned from start to end. Find becomes *O(n)*. Keeping the sets in sorted arrays speeds up the search, but every union requires merging or re-sorting two sorted arrays. The cost of that goes up to *O(n lg n)*.

Method	Union	Find	Problem
Linked list	O(1)	O(n)	Slow search
Keep sorted	—	—	Sorting O(n lg n)
Parent array (tree)	O(1)	Tree depth	Depth must be kept small

Table 16.2: Three ways to implement disjoint sets.

The third row is the most promising. Union is again just writing a single link. The cost of find, on the other hand, equals the depth of the tree. If the depth can be kept small, find becomes fast as well.

Trees are usually built from node structures and pointers. That is not needed here. The items are already numbered from *0* to *n − 1*; the number can be used directly as an array index. Each node needs only a single piece of information: the number of its parent. So the whole forest fits into a single integer array of *n* elements. It is possible to implement it with the array, the most reliable, robust and fastest data structure. Access to an item in an array is *O(1)*, no pointer chasing is needed, `malloc` is called just once, and since memory is contiguous the cache is also used well.

16.2.2 The parent array

Definition 16.2

A **parent array** is an `int` array of *n* elements. `parent[i]` is the number of the parent of node *i* in the tree, that is, of the node one level above it. If *i* is a root, `parent[i]` holds a negative value. In the first version this value is −1.

The node immediately above a node is its parent; the `parent` array in the code holds these parent links. Since a root has no parent, no valid index can be written into its cell. Because node numbers are 0 or positive, a negative value cannot be confused with any node and is safely used as the "this is a root" mark. Figure 16.3 shows the parent array of the example forest.



Figure 16.3: The parent array of Example 16.1. `parent[i]` is the parent of *i*; roots hold −1.

The array is easy to read. The parent of 1 and 9 is 4; `parent[1]` and `parent[9]` contain 4. The parent of 3 and 5 is 2. The parent of 6, 7 and 8 is 0. 0, 2 and 4 are the roots of the sets: `parent[0] = parent[2] = parent[4] = -1`. In C this array is defined as follows:

```
#define N 10
int parent[N] = { -1, 4, -1, 2, -1, 2, 0, 0, 0, 4 };
```

In memory this array is 10 consecutive `int` cells. On a 64-bit Windows or Linux machine an `int` is 4 bytes and the array takes 40 bytes. If the array starts at address 0x1000, the cell `parent[i]` is at address *0x1000 + 4i*. Table 16.3 gives this view. In a tree with pointers each node would be a separate `malloc` block and carry at least one pointer (8 bytes). Here 4 bytes per node are enough and all the trees sit in a single block.

Address	0x1000	0x1004	0x1008	0x100C	0x1010	0x1014	0x1018	0x101C	0x1020	0x1024
Index	0	1	2	3	4	5	6	7	8	9
Value	−1	4	−1	2	−1	2	0	0	0	4

Table 16.3: The `parent` array in memory (the array is assumed to start at 0x1000; addresses vary by machine).

The parent array defines a tree completely by itself. Given the array, the tree can be drawn: for each *i* it is enough to draw an arrow from *i* to `parent[i]`. The reverse is also true: every tree gives a single parent array. However, the array does not say directly what the children of a node are. To find the children of 0, one must scan the whole array and collect the cells whose value is 0. Since union-find never asks this question, this shortcoming is not a problem.

16.2.3 Building the parent array in dynamic memory

The code in the slides defines the `parent` array as a global array of size `#define N 10`. This keeps the routines short and is enough for examples. In a real program, however, the number of items is usually known only at run time: the number of vertices of a graph read from a file is not known at compile time. In that case the array is allocated with `malloc`. The following code gathers the number of items and the array into a single structure:

```
typedef struct {
  int  n;      /* number of items */
  int *parent;    /* parent[i]: parent of i; at a root, -(number of nodes) */
} sets;

sets *sets_create(int n) {
  sets *k = malloc(sizeof *k);
  if (k == NULL) return NULL;            /* out of memory */
  k->parent = malloc(n * sizeof *k->parent);
  if (k->parent == NULL) {                  /* array could not be allocated */
    free(k);                              /* give back the half-built structure */
    return NULL;
  }
  k->n = n;
  for (int i = 0; i < n; i++)
    k->parent[i] = -1;                      /* every item is a singleton set */
  return k;
}

void sets_free(sets **sp) {
  if (*sp == NULL) return;
  free((*sp)->parent);                      /* inner array first */
  free(*sp);                             /* then the structure itself */
  *sp = NULL;                            /* the caller's pointer becomes empty */
}
```

`sets_create` allocates two separate blocks. The first is the structure itself: an `int` and a pointer. On a 64-bit machine `sizeof(sets)` is 16 bytes: 4 bytes for `n`, 4 bytes of alignment padding and 8 bytes for the pointer. The second block is the parent array of *n* integers. In the expression `malloc(n * sizeof *k->parent)`, `sizeof *k->parent` gives the size of the array's element type. Even if the type is later changed to `long`, this line does not have to change.

After every `malloc` call a `NULL` check is made. If the first block cannot be allocated, there is nothing to do; the routine returns `NULL`. If the second block cannot be allocated, the situation is subtler. At this point there is a block that was allocated for the structure. If the routine simply returned `NULL`, that block would be in nobody's hands and could never be given back. This is a **memory leak**. That is why the code first frees the half-built structure with `free(k)` and then returns `NULL`. The last loop makes every item a singleton set: at the start all cells are −1.

`sets_free` gives the blocks back in the reverse of the order they were allocated. First the inner array, then the structure is released. If the order were reversed, that is, if `free(*sp)` were called first, the expression `(*sp)->parent` would read a block that had been given back. This is undefined behavior. The routine takes a **pointer to pointer** as its parameter: `sets **sp`. That way it can write `NULL` into the caller's own variable in the last line. After the caller says `sets_free(&k)`, `k` is `NULL`. If `sets_free(&k)` is called a second time by mistake, nothing happens because of the check in the first line. If the parameter were just `sets *k`, the routine could set its own copy to `NULL`, while the caller's `k` would keep pointing to the block that had been given back. Such a pointer is called a **dangling pointer**.

These two routines were tried with a counting `malloc` / `free` wrapper: at the end of creation and deletion, 2 allocations and 2 frees were counted, and no leak was seen. In the rest of the chapter, to stay consistent with the slides, the routines use the global `parent` array. To move to the dynamic structure, a parameter `sets *k` is added to each routine and `k->parent[i]` is written in place of `parent[i]`; the logic does not change at all.

16.2.4 The first version: `find1` and `union1`

The first version is the most direct C translation of the model:

```
int find1(int i) {
  for (; parent[i] >= 0; i = parent[i])
    ;                /* climb until negative */
  return i;
}
void union1(int i, int j) {
  parent[i] = j;        /* hook set i under j */
}
```

`find1` is a single `for` loop. The initialization part of the loop is empty, because the starting node is already the parameter `i`. The condition `parent[i] >= 0` says "*i* is not a root". While the condition is true, `i = parent[i]` climbs to the next node up. The body of the loop is empty; the lone semicolon shows this. All the work is done in the condition and update parts. The loop stops at the first node whose parent value is negative, that is, at the root. The routine returns this root.

A loop with an empty body is a common idiom in C, but it is not easy to read. Writing the semicolon on a separate line with a comment next to it is a deliberate choice. Had the semicolon been written right at the end of the `for` line, a reader might have taken it for a typo.

`union1` is a single assignment: `parent[i] = j`. The parent of the root *i* is now *j*; the whole tree of *i* goes under *j*. The routine has a **precondition**: *i* and *j* must be the roots of different sets. If an item that is not a root is given, its root is first found with `find1`. If the precondition is violated, either a tree is split in two or a root points to itself and the find routine falls into an infinite loop.

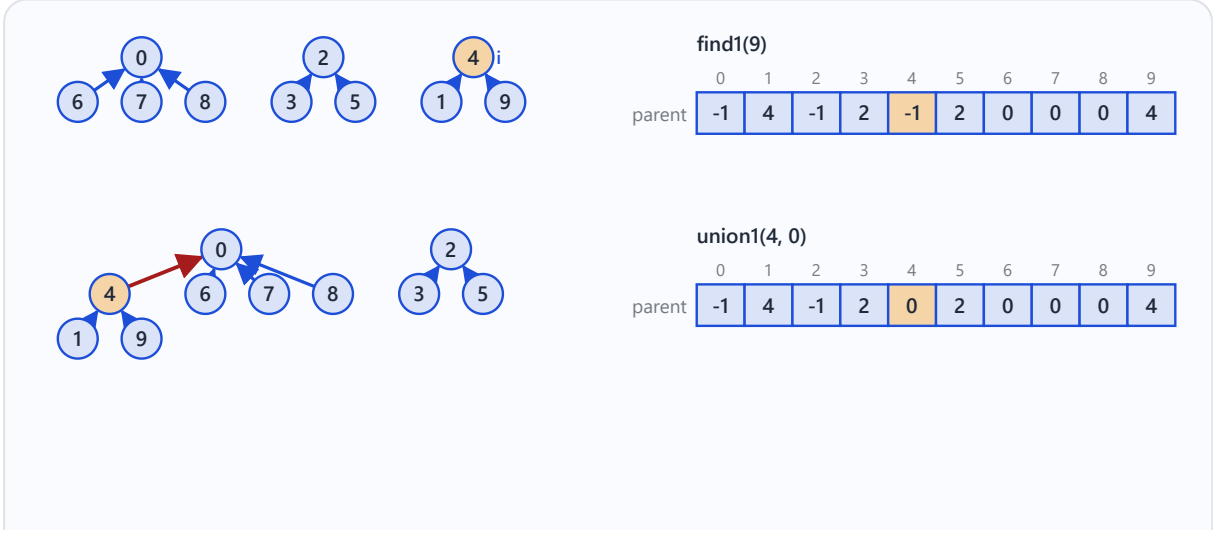
The costs are clear. `union1` makes a single assignment: $\Theta(1)$. `find1(i)` runs the loop once for each edge on the path from *i* to the root. If the distance of *i* from the root is *d*, it makes *d* hops and *d + 1* comparisons: $\Theta(d + 1)$. In this chapter a "hop" means one execution of the assignment `i = parent[i]`. In the worst case *d* is the distance of the deepest node of the tree from the root. The extra memory of both routines is $O(1)$: only the variable `i` is used.

16.2.5 Code trace

In the example forest, let `find1(9)`, `union1(4, 0)` and again `find1(9)` be called in this order. Table 16.4 gives the state of `i` and of the array at each step. Figure 16.4 draws three important moments.

Step	Operation	i	Check	Result
1	`find1(9)`	9	`parent[9] = 4 ≥ 0`	`i = 4`
2		4	`parent[4] = −1 < 0`	Root; `return 4`
3	`union1(4, 0)`	—	—	`parent[4] = 0`
4	`find1(9)`	9	`parent[9] = 4 ≥ 0`	`i = 4`
5		4	`parent[4] = 0 ≥ 0`	`i = 0`
6		0	`parent[0] = −1 < 0`	Root; `return 0`

Table 16.4: Trace of `find1(9)`, `union1(4, 0)`, `find1(9)`. Initially `parent = [-1, 4, -1, 2, -1, 2, 0, 0, 0, 4]`.



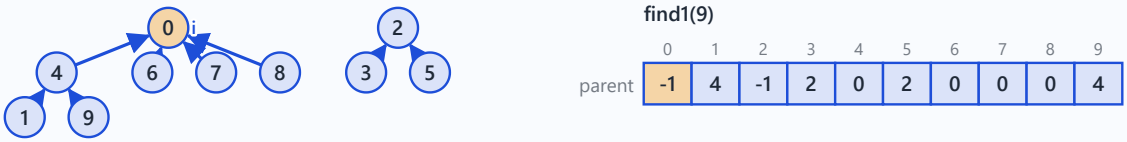


Figure 16.4: Three moments of the code trace. Top: the first `find1(9)` stops at 4. Middle: `union1(4, 0)` changes a single cell: `parent[4] = 0`. Bottom: the second `find1(9)` walks one hop further and stops at 0.

The first call reaches 4 with a single hop: 9 is in set number 4. The union changes just one cell. But this single assignment increases by one the distance to the root of every node below 4. The second find call therefore makes two hops and returns 0: now 9 is in set number 0. The cost of find equals the depth of the node searched for. Union is cheap, but its price shows up in later find operations.

The values in the table were verified with a driver compiled together with the code in the slides: the first call returned 4 and the second returned 0. In the same driver the comparison `find1(8) == find1(6)` also gave 1 (true).

16.3 Analysis of the first version

It is not enough for an algorithm to be correct. It must be efficient and written according to the rules of software engineering. Even these are not enough: one must ask which data will stress the algorithm the most. Whatever happens in the background, what is the longest the code can run? Asking and answering this question is what separates someone who knows a programming language from a computer engineer. If an employer says “you all know C, so you all get the same salary”, the engineer's answer is ready: give a problem, and look at how it is solved and how long the solution takes in the worst case.

16.3.1 A bad sequence of operations

Let us start with n singleton sets: $S_i = \{i\}$, with `parent[i] = -1` for every i. Then let the following sequence of operations be applied:

```
/* start with n singleton sets */
union1(0, 1);
union1(1, 2);
...
union1(n-2, n-1);
find1(0);
find1(1);
...
find1(n-1);
```

Each union attaches the root of the tree built in the previous step under a new singleton set. `union1(0, 1)` puts 0 under 1. `union1(1, 2)` puts 1, and with it the tree made of 0 and 1, under 2. Both parameters of every union really are roots; the precondition is not violated. But the tree grows by one level at every step. Figure 16.5 shows the result for $n = 5$.



Figure 16.5: For $n = 5$, after four `union1` calls the tree is a chain (left): `parent = [1, 2, 3, 4, -1]`. On the right, `find1(0)` has made its fourth hop and reached the root.

Each of the unions takes constant time. The problem appears in the find operations. Table 16.5 gives, for each call from `find1(0)` to `find1(4)` on the chain, how many hops it makes.

Call	Level of the node	Path followed	Hops
`find1(0)`	5	0 → 1 → 2 → 3 → 4	4
`find1(1)`	4	1 → 2 → 3 → 4	3
`find1(2)`	3	2 → 3 → 4	2
`find1(3)`	2	3 → 4	1
`find1(4)`	1	4	0
Total			10 = 5 · 4 / 2

Table 16.5: Find calls on the chain with $n = 5$. The level is the position when the root is counted as level 1.

16.3.2 Total cost

The pattern in the table is the same for every n. In the chain, node i is at level $(n - i)$. It is $n - 1 - i$ edges away from the root.

Theorem 16.1 (Worst case of the first version)

Apply the calls `union1(i, i+1)` for $i = 0, 1, \ldots, n - 2$ in this order to n singleton sets. Then call `find1` once for each item. The total number of hops is $n(n - 1)/2$. The cost of this sequence of operations is $\Theta(n^2)$.

Proof. Each of the $n - 1$ unions makes one assignment; their total is $\Theta(n)$. After the unions, `parent[i] = i + 1` ($i < n - 1$) and `parent[n-1] = -1`. So `find1(i)` passes through the nodes $i, i + 1, \ldots, n - 1$ and makes $n - 1 - i$ hops. The total is $\Sigma_{i=0}^{n-1} (n - 1 - i) = (n - 1) + (n - 2) + \ldots + 1 + 0 = n(n - 1)/2$. The total cost is $\Theta(n) + \Theta(n^2) = \Theta(n^2)$. ■

A single `find1` takes *O(n)* in the worst case; *n* of them climb to $O(n^2)$ in total. This is unacceptable in problems with millions of items where union-find is used. The measurement in Table 16.6 makes the growth concrete. When *n* doubles, the number of hops and the time grow about fourfold; this is the sign of quadratic growth.

n	Hops (first version)	*n(n − 1)/2*	Time	Hops (weighted)
1.000	499.500	499.500	≈ 0,001 s	999
10.000	49.995.000	49.995.000	≈ 0,13 s	9.999
20.000	199.990.000	199.990.000	≈ 0,5–0,6 s	19.999
40.000	799.980.000	799.980.000	≈ 2,0–2,7 s	39.999

Table 16.6: Measurement of *n* find calls on the chain (`zig cc -O2` , x86-64; times vary by machine). The last column is the same union requests done with the weighting rule.

This sequence of operations spoils the structure of the tree. But the union operation itself is not the culprit. The real problem is which tree is attached under which in the union. At every step the large tree (the chain) was attached under the small single-node tree. Had the order been reversed, that is, had each new node been attached to the root of the large tree, the tree would always have stayed two levels deep. The last column of the table shows this: when the same union requests are made with the weighting rule, the tree stays at two levels and the total of *n* finds is only *n − 1* hops. The weighting rule rests exactly on this idea.

16.3.3 Testing the worst case

In worst-case analysis, the code must be tested with the data on which it will crash or run the longest. These data are hard to identify. The sequence of unions that produces a chain is exactly such an input for `find1` . Random tests easily miss it. When the roots of two randomly chosen items are united, the trees do not grow long like chains. In the measurement, 200,000 random operations were performed on *n = 100,000* items; each operation made two finds, and half of the operations included a union. The first version made an average of 1.108 hops per find, and the deepest node was at level 10,130. This is a bad result, but it is still very far from the *n* levels of a chain. Someone who tests a program only with random data may never see the real worst case.

What the software world calls a “beta version” is an open admission of this difficulty. The producer tells the user: there may be errors we overlooked; if you find any, report them so we can fix them. Software testing is a field in its own right, and many papers have been written about it. The lesson of this chapter is simpler: when a routine is written, one must think of the input that will stress it most and test it with that input as well.

16.4 The weighting rule

16.4.1 The rule

In a union, the root of one tree goes under the other. The distance to the root of every node of the tree that goes under increases by 1. The nodes of the other tree stay where they are. To keep the cost low, the number of nodes whose distance increases should be small. So the tree with fewer nodes should be attached under the tree with more nodes.

Definition 16.3

Weighting rule: when performing `union(i, j)`, if the tree of *i* has fewer nodes than the tree of *j*, then *j* is made the parent of *i*; otherwise *i* is made the parent of *j*. In short, the root with more nodes becomes the parent. This rule is also called **union by size**.

Figure 16.6 shows the effect of the rule. A tree with five nodes rooted at 0 and a tree with two nodes rooted at 5 are merged. In the union on the left, the small tree goes under the large one; only the distance of 5 and 6 from the root grows, and the tree stays at 3 levels. In the union on the right, the large tree goes under the small one; the distance of five nodes grows at once, and the tree climbs to 4 levels.



Figure 16.6: Two unions of the same two trees. Left: the small tree is attached to the large one: depth 3. Right: the large tree is attached to the small one: depth 4.

To apply the rule, the number of nodes of each tree must be known. Should a separate array be kept for that? There is no need. For all nodes other than the root, the parent array holds 0 or positive values. Roots, on the other hand, hold a negative value just to say "I am a root", and the magnitude of that value has so far never been used. So the number of nodes of a root can be stored in the parent array with a negative sign. At the root, −(number of nodes) is written instead of −1. The sign carries the "root" information and the absolute value the "size" information.

16.4.2 `union2`: minus the node count at the root

```
void union2(int i, int j) {
  /* weighting rule; i and j must be roots */
  int p = parent[i] + parent[j];  /* -(total nodes) */
  if (parent[i] > parent[j]) {    /* i has fewer nodes */
    parent[i] = j;              /* hook i under j */
    parent[j] = p;              /* update j's count */
  } else {                    /* j has fewer or equal */
    parent[j] = i;
    parent[i] = p;
  }
}
```

The routine works line by line as follows:

- **Line 3.** `p = parent[i] + parent[j]`. The values at the two roots are negative node counts. Their sum is the negative of the node count of the merged tree. For a tree with 3 nodes and a tree with 5 nodes, $p = -3 + (-5) = -8$.
- **Line 4.** The comparison `parent[i] > parent[j]` looks backward at first glance. Because the values are negative, the larger value indicates the smaller size: $-2 > -4$ says that 2 nodes are fewer than 4 nodes. If the condition is true, the tree of i has fewer nodes.
- **Lines 5–6.** Since i is smaller, it is hooked under j: `parent[i] = j`. The new root is j, and the merged node count is written into its cell: `parent[j] = p`.
- **Lines 7–10.** Otherwise, that is, if j has fewer or an equal number of nodes, j goes under i and the cell of i becomes p. In case of a tie, attaching j is an arbitrary choice; both choices satisfy the rule.

The order of the assignments matters. If one of the two cells were changed before p is computed in line 3, the node count information would be lost. For example, if `parent[i] = j` were written first, `parent[i]` would no longer hold the node count of i but the number of j. That is why the routine takes the sum into a local variable at the very beginning.

The cost is still $\Theta(1)$: one addition, one comparison and two assignments. No extra array is needed; the node counts are stored in what was an unused information field. That all cells start at −1 also fits the new meaning: a singleton set has one node.

`find1` stays unchanged. All it cares about is whether the value is negative; it makes no distinction between −1 and −8. This is a nice consequence of the choice of representation: the extra information stored at the root does not affect the find routine at all.

16.4.3 Code trace

Let us start with eight singleton sets. First, adjacent pairs are united: `union2(0, 1)`, `(2, 3)`, `(4, 5)`, `(6, 7)`. Then the two-node trees two by two: `(0, 2)`, `(4, 6)`. Finally the two four-node trees: `(0, 4)`. In every union the two trees are of equal size. Table 16.7 gives the values at each step.

Call	`parent[i]`, `parent[j]`	`p`	`parent` afterwards	Level
`(0, 1)`	−1, −1	−2	`[-2, 0, -1, -1, -1, -1, -1, -1]`	2
`(2, 3)`	−1, −1	−2	`[-2, 0, -2, 2, -1, -1, -1, -1]`	2
`(4, 5)`	−1, −1	−2	`[-2, 0, -2, 2, -2, 4, -1, -1]`	2
`(6, 7)`	−1, −1	−2	`[-2, 0, -2, 2, -2, 4, -2, 6]`	2
`(0, 2)`	−2, −2	−4	`[-4, 0, 0, 2, -2, 4, -2, 6]`	3
`(4, 6)`	−2, −2	−4	`[-4, 0, 0, 2, -4, 4, 4, 6]`	3
`(0, 4)`	−4, −4	−8	`[-8, 0, 0, 2, 0, 4, 4, 6]`	4

Table 16.7: Trace of seven `union2` calls. In every row the condition is false (the `else` branch): first `parent[j] = i`, then `parent[i] = p` is written.

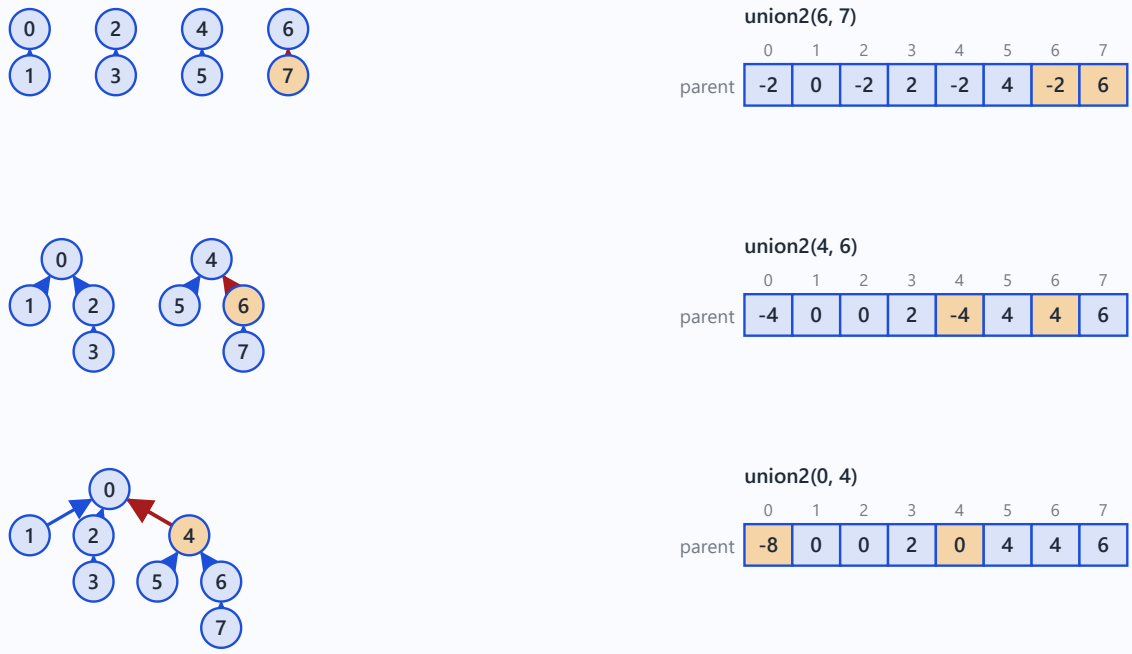


Figure 16.7: Three moments of the trace. Top: four trees with two nodes. Middle: two four-node trees after `union2(4, 6)`. Bottom: a single tree after `union2(0, 4)`: 8 nodes, 4 levels, −8 at the root.

In every row the condition is false, because the values at the two roots are equal: none of the expressions $-1 > -1$, $-2 > -2$ and $-4 > -4$ is true. So every time the `else` branch runs and j goes under i. The number of levels grows only when two trees of equal size are united: 2, 3 and 4. The tree with eight nodes has 4 levels. This is exactly the bound given by Theorem 16.2: $\lfloor \lg 8 \rfloor + 1 = 4$.

In the final state the deepest node is 7: the path 7 → 6 → 4 → 0 takes three hops. All the arrays in the table are identical to the output of the driver compiled with the `union2` code in the slides.

16.4.4 The depth lemma

How badly can the weighting rule distort the tree? The sequence of operations in the trace is the worst case for this rule: every time two trees of equal size are united. In each union the depth of all nodes of the small tree grows by 1, but the tree grows at least twofold. The second observation bounds the first.

Theorem 16.2 (Depth under weighted union)

Let T be a tree with m nodes obtained by starting from single-node trees and using only `union2`. The number of levels of T cannot exceed $\lfloor \lg m \rfloor + 1$.

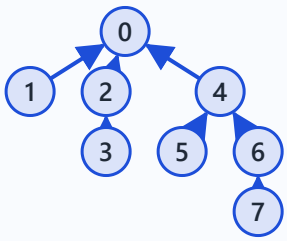
Proof. We prove a stronger statement: every tree with d levels has at least 2^{d-1} nodes. The proof is by induction on the number of unions performed.

If no union has been performed, the tree has a single node: $d = 1$ and $1 \geq 2^0$. Now suppose that two trees with node counts $m_a \geq m_b$ and level counts d_a, d_b satisfy the statement. `union2` attaches the tree b to the root of a (in a tie, too, one of the trees is attached; the naming is chosen accordingly). The number of levels of the new tree is $d = \max(d_a, d_b + 1)$ and its number of nodes is $m = m_a + m_b$. There are two cases:

- If $d = d_a$, then $m \geq m_a \geq 2^{d_a - 1} = 2^{d-1}$.
- If $d = d_b + 1$, then $m = m_a + m_b \geq 2m_b \geq 2 \cdot 2^{d_b - 1} = 2^{d-1}$.

In both cases the statement is preserved. Hence $m \geq 2^{d-1}$, that is, $d - 1 \leq \lg m$. Since $d - 1$ is an integer, we get $d - 1 \leq \lfloor \lg m \rfloor$ and $d \leq \lfloor \lg m \rfloor + 1$. ∎

The essence of the proof is this. The depth of a node grows by 1 only when its tree is attached to a tree that is smaller or equal. At that moment the tree containing the node grows at least twofold. With m nodes, the size can double at most $\lg m$ times. So the depth of no node can grow by more than $\lg m$.



m = 8 → 4 levels = ⌊lg 8⌋ + 1

Figure 16.8: The tree with 8 nodes built from equal pairs reaches the bound exactly: 4 levels = $\lfloor \lg 8 \rfloor + 1$.

The bound is tight, that is, nothing better can be said. A tree built from equal pairs reaches it exactly for every $m = 2^k$. In the measurement, 5 levels were obtained with 16 nodes and 21 levels with $2^{20} = 1{,}048{,}576$ nodes; both equal $\lfloor \lg m \rfloor + 1$. With random unions the trees stay far below the bound: with 10^6 items the bound is 20, while the deepest node was at level 9.

The result is a big gain. The worst-case cost of a single find dropped from $O(n)$ to $O(\lg n)$. With a million items this means at most 19 hops instead of a million. The total cost of a sequence of $n - 1$ unions and m finds is $O(n + m \lg n)$. In a mathematics course the question would continue here with "prove that". For an engineer, what matters is that the complexity dropped from n to $\lfloor \lg n \rfloor + 1$. This is what improving an algorithm means. But we do not stop here; one more step can be taken.

Common mistake

Reading the comparison of negative values backward. If `if (parent[i] < parent[j])` is written, the condition means "i has more nodes", and the large tree is attached under the small one. The code compiles, the sets stay correct and the results are right; only the trees turn into chains and the cost silently returns to $O(n)$. Mistakes of this kind are noticed only with a worst-case input or by measuring the tree depth.

16.4.5 Uniting any two items

`union1` and `union2` require roots. In practice, requests come such as "put 3 and 8 into the same set", and 3 and 8 may not be roots. A small wrapper routine can fill this gap:

```
int unite(int x, int y) {   /* x and y are any two items */
  int a = find2(x);         /* root of x */
  int b = find2(y);         /* root of y */
  if (a == b) return 0;          /* already in the same set */
  union2(a, b);              /* unite the two roots */
  return 1;
}
```

The routine first finds the roots of the two items. If the roots are the same, the items are already in the same set and nothing is done; the return value is 0. If they differ, the two roots are given to `union2` and 1 is returned. The return value answers the caller's question "did this request perform a new union?" This information is used directly in counting components and in Kruskal's algorithm. The `find2` used in the routine is the version of find that applies the collapsing rule (Definition 16.4); `find1` can be used in its place as well. The cost equals the cost of two finds; the contribution of the union is constant.

The "are the roots the same?" check is a safety measure. If `union1(r, r)` is called with the same root, `parent[r] = r` results: the root points to itself. Every `find1` that follows loops forever at this node, because the condition `parent[r] ≥ 0` is never false. In the experiment the loop was cut off

after a million steps. `union2(r, r)` breaks in a different way: $p = 2 \cdot parent[r]$ is computed, the condition comes out false, first `parent[r] = r` and then `parent[r] = p` are written. In the end the root remains a root, but its node count appears to have doubled. In a three-item experiment the array after `union2(1, 1)` was `[-1, -2, -1]`: a singleton set is believed to have 2 items. This wrong count misleads the weighting rule in later unions.

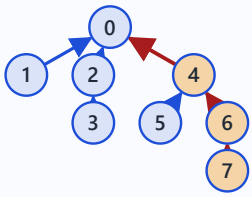
16.5 The collapsing rule

16.5.1 The rule

In the eight-node tree built with the weighting rule (the lower part of Figure 16.7), `find2(7)` reaches the root in three hops. If the program calls `find2(7)` again and again, every call walks the same three steps anew and the cost multiplies. Attaching the tree with fewer nodes to the one with more nodes bounds the depth by $lg\ n$, but it does not completely prevent the depth from growing. Since we can no longer check the depth at union time, another opportunity is sought: a find operation walks the whole path anyway. While walking, it can shorten the path.

Definition 16.4

Collapsing rule: during `find2(i)`, if i is not a root, every node j on the path from i to the root is attached directly to the root. The general name of this idea is **path compression**.



every node on the path 7 → 6 → 4 → 0 will be attached to t

Figure 16.9: During `find2(7)`, every node on the path 7 → 6 → 4 → 0 will be attached directly to 0.

The rule takes advantage of the “meaningless” placement in trees. The position of an item in the tree carries no information; all that matters is which root it reaches. Whether 7 sits under 6 or directly under 0 does not change the sets. What changes is only the cost of later finds. In graph theory the same idea goes by the name “closure”: an indirect connection is shortened by a direct edge. Although the problems are different, the underlying structure is the same, and this similarity is often useful in graph problems.

16.5.2 `find2`

```
int find2(int i) {
  int r, s;
  for (r = i; parent[r] >= 0; r = parent[r])
    ;                   /* find the root */
  while (i != r) {      /* collapse the path */
    s = parent[i];         /* save i's parent */
    parent[i] = r;         /* hook i directly to the root */
    i = s;              /* move on to its parent */
  }
  return r;
}
```

The routine passes over the path twice.

- **First pass (lines 2–4).** The `for` loop does what `find1` does, but it walks with a separate variable `r` without changing `i`. When the loop ends, `r` is the root. `i` still refers to the starting node; the second pass will start from there.
- **Second pass (lines 5–9).** The `while` loop walks again from `i` up to the root. Three things are done in each round. First, `s = parent[i]` saves the old parent of `i`. Then `parent[i] = r` attaches `i` directly to the root. Finally `i = s` moves on to the saved parent. The loop ends when `i` reaches the root.
- **Line 10.** The root is returned.

The variable `s` is the vital part of the routine. The assignment `parent[i] = r` erases the old parent of `i`. If the old parent is not saved beforehand, the rest of the path is lost; the loop cannot move on to the next node. This is exactly like having to save the next node when reversing a linked list: before a link is changed, the place that link points to must be kept in a variable.

The second loop has a small flaw. It also runs once for a node that is already attached directly to the root and, with the assignment `parent[i] = r`, rewrites the same value into the cell. This is unnecessary but harmless. Since the last node of the path is always attached to the root, exactly one "extra round" is made in every call except for the root itself.

Cost analysis: if the distance of the starting node from the root is d, the first loop runs d times and the second loop also d times. The cost of a single call is $\Theta(d + 1)$; that is, about twice that of `find1` alone. Together with the weighting rule, $d \le \lfloor \lg n \rfloor$, so a single call is $O(\lg n)$ in the worst case. The gain is seen not in a single call but in the sequence of calls: all the nodes on the path are now one hop from the root. The extra memory is $O(1)$: `r` and `s`.

The negative number at the root is not disturbed by path compression. Only the cells of nodes other than the root change, and the root's cell is never written to. The number of nodes in the tree does not change either, because no node leaves the tree. The weighting rule therefore works smoothly together with path compression.

16.5.3 Code trace

Let us start with the tree at the end of the weighted trace: `parent = [-8, 0, 0, 2, 0, 4, 4, 6]`. Let `find2(7)`, then again `find2(7)` and `find2(5)` be called in this order. Table 16.8 gives the first call step by step.

Step	Loop	State	Operation	Changed cell
1	`for`	`r = 7`	`parent[7] = 6 ≥ 0` → `r = 6`	—
2	`for`	`r = 6`	`parent[6] = 4 ≥ 0` → `r = 4`	—
3	`for`	`r = 4`	`parent[4] = 0 ≥ 0` → `r = 0`	—
4	`for`	`r = 0`	`parent[0] = -8 < 0` : root	—
5	`while`	`i = 7`	`s = 6`, `parent[7] = 0`, `i = 6`	`parent[7]` : 6 → 0
6	`while`	`i = 6`	`s = 4`, `parent[6] = 0`, `i = 4`	`parent[6]` : 4 → 0
7	`while`	`i = 4`	`s = 0`, `parent[4] = 0`, `i = 0`	(already 0: extra round)
8	—	`i == r`	`return 0`	`parent = [-8, 0, 0, 2, 0, 4, 0, 0]`

Table 16.8: Trace of `find2(7)`. Initially `parent = [-8, 0, 0, 2, 0, 4, 4, 6]`.

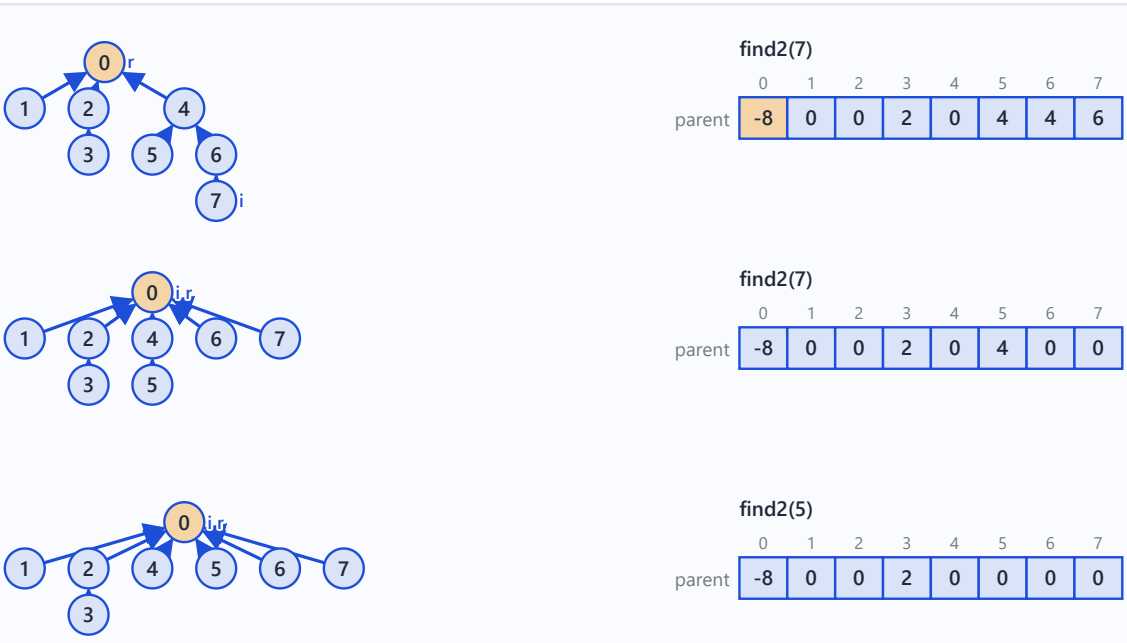


Figure 16.10: Top: the end of the first pass: `r = 0` has been found, and the tree has not changed yet. Middle: `find2(7)` is finished: 7 and 6 are attached directly to the root. Bottom: after `find2(5)` : 5 is attached to the root too; in the tree only 3 remains under 2.

The second call `find2(7)` is short. Since `parent[7] = 0`, the first loop finds the root in a single hop. The second loop also runs one round; this round is again an “extra” round, because 7 is already attached to the root. The array does not change.

`find2(5)`, on the other hand, opens up a new path: 5 → 4 → 0. The first loop makes two hops. The second loop attaches 5 to the root (`parent[5]`: 4 → 0), then runs an extra round for 4. The final state is `parent = [-8, 0, 0, 2, 0, 0, 0, 0]`. Six of the eight nodes are now attached directly to the root; only 3 is two hops away. All the arrays were compared with the output of the driver compiled with the code in the slides.

> **Common mistake**
>
> Attaching without saving the parent in the `while` loop: `parent[i] = r; i = parent[i];`. The second assignment now reads `r ; i` jumps straight to the root and the rest of the path is left uncompressed. Since the routine returns the right root, the error does not show in the output; only the expected speedup does not come. The correct way is to take the old parent into `s` first.

16.5.4 How is this an improvement?

At first glance the objection is fair: there was one `for` loop, and a `while` loop has been added. How can this be an improvement? The answer is seen by looking not at a single call but at a sequence of calls. When `find2(7)` is called a second time, and on every later call, it finds the root in one stroke. Table 16.9 compares the cost of the two versions on the same tree.

Call	`find1` (hops)	`find2` (hops + linking)
1st `find2(7)`	3	3 + 3
2nd `find2(7)`	3	1 + 1
10 times `find2(7)`	30	6 + 9 · 2 = 24
`find2(5)` (afterwards)	2	2 + 2, then 1 + 1

Table 16.9: Cost of the two find versions on the same tree (`parent = [-8, 0, 0, 2, 0, 4, 4, 6]`). For `find2`, "hops + linking" is written.

The first call is about twice as expensive: 6 steps instead of 3. The second call takes 2 steps instead of 3. Over ten calls the total is 24 against 30, and the difference grows with the number of calls. In a thousand calls `find1` makes 3,000 hops, while `find2` stays at $6 + 999 \cdot 2 = 2{,}004$ steps. The price of shortening the path is paid once, and its return is collected on every later call.

This kind of calculation is called **amortized analysis**. Instead of the worst-case cost of a single operation, the total cost of a long sequence of operations is divided by the number of operations. Expensive operations happen once in a while and make the operations that follow them cheaper. When the cost is spread over many operations, it drops to nearly constant per operation.

The effect of path compression is seen most clearly on the worst tree of the weighting rule. On the tree with 2^{20} nodes built from equal pairs (21 levels), 2^{20} random finds were made. With the weighting rule alone an average of 10.00 hops per find was measured; since the tree does not change,

this number never decreases. With path compression it dropped to an average of 1.78 hops, and the number of levels of the tree fell from 21 to 6. The time also fell from 0.18 seconds to 0.03 seconds (varies by machine).

16.5.5 Recursive path compression

Path compression can also be written in a single line with recursion:

```
int find(int i) { return parent[i] < 0 ? i : (parent[i] = find(parent[i])); }
```

It reads as follows. If *i* is a root (`parent[i] < 0`), it is returned itself. Otherwise the root of its parent is first found recursively: `find(parent[i])` . The returned root is written into `parent[i]` and the same value is returned. The value of an assignment expression is the assigned value; so `(parent[i] = find(parent[i]))` both updates the link and yields the root.

Recursion compresses the path from the root backward. The calls descend to the root, and on the way back each frame attaches its own node to the root. On the eight-node example tree, `find(7)` calls `find(6)` , which calls `find(4)` , which calls `find(0)` . `find(0)` returns 0. On the way back `parent[4] = 0` , `parent[6] = 0` and `parent[7] = 0` are written in turn. The result is the same as with `find2(7)` : `[-8, 0, 0, 2, 0, 4, 0, 0]` . After that, `find(5)` gives `[-8, 0, 0, 2, 0, 0, 0, 0]` , which is also the same.

The difference between the two versions concerns memory. The looping `find2` uses *O(1)* extra memory. The recursive `find` opens as many call frames as the length of the path: *O(d)*. With the weighting rule $d \le \lg n$, so this is not a problem; even with a billion items at most 30 frames are opened. But in a chain built with unweighted union, *d* can reach *n* and the call stack can overflow.

16.6 The Tarjan–van Leeuwen result and the Ackermann function

16.6.1 Almost constant cost

What is the cost when the two rules are used together? The weighting rule bounds a single find by *O(lg n)*. The collapsing rule, since it shortens paths, makes repeated finds cheaper. It is not easy to compute exactly the combined effect of the two. The question was closed by R. E. Tarjan and J. van Leeuwen in 1984.

Theorem 16.3 (Tarjan and van Leeuwen)

Start with n singleton sets. Let f finds and u unions be performed in any mixed order, using the weighting rule and the collapsing rule. Let the maximum time required be $T(f, u)$. For $u \geq n/2$ there are positive constants k_1 and k_2 such that

$$k_1 \cdot (n + f \cdot \alpha(f + n, n)) \leq T(f, u) \leq k_2 \cdot (n + f \cdot \alpha(f + n, n))$$

holds. Here α is the inverse Ackermann function.

The theorem says two things. The upper bound shows that the total cost of the operations is proportional to $n + f \cdot \alpha$. The lower bound says that nothing better than this is possible with these two rules; the analysis is tight. The proof of the theorem is beyond the scope of this chapter. But to interpret the result it is enough to know how slowly α grows. α grows so slowly that its value does not exceed 4 even for as many items as there are atoms in the universe. When the total cost is divided by the number of operations, the cost per operation comes out as nearly $O(1)$.

Consider a million items. One program finds the set of an item in nearly constant time, while another still scans lists one by one. Both give the correct result, but one finishes in a small fraction of a second and the other runs for minutes. This is also the difference between a mathematician and an engineer facing the same job: besides the correct result, the engineer also asks about the cost.

16.6.2 The Ackermann function

To understand α, one must first look at the function of which it is the inverse. The Ackermann function is a recursively defined function of two natural-number arguments that grows extremely fast.

Definition 16.5

The **Ackermann function** is defined for $m, n \geq 0$ as follows:

```
/* Ackermann: grows very fast */
A(m, n) = n + 1                 m = 0
        = A(m-1, 1)             m > 0, n = 0
        = A(m-1, A(m, n-1))     m > 0, n > 0
```

The definition has three cases. If $m = 0$, the result is $n + 1$; the recursion ends here. If $n = 0$, the first argument decreases by one and the second argument becomes 1. The third case is the most interesting: the second argument of the outer call is the result of an inner call. The inner call is $A(m, n - 1)$ and the outer call is $A(m - 1, \ldots)$. At every step either m shrinks, or m stays the same and n shrinks. So the recursion always terminates; but when it terminates is another question.

	n = 0	n = 1	n = 2	n = 3	n = 4
m = 0	1	2	3	4	5
m = 1	2	3	4	5	6
m = 2	3	5	7	9	11
m = 3	5	13	29	61	125
m = 4	13	65533	$2^{65536} - 3$	...	...

Table 16.10: Values of *A(m, n)*. The first four rows were computed with a recursive C program and verified.

Each row is an iteration of the previous one, and so has a closed form. $A(1, n) = n + 2$, $A(2, n) = 2n + 3$ and $A(3, n) = 2^{n+3} - 3$. In the row $m = 3$ the values grow exponentially. In the row $m = 4$ a tower of exponents begins: $A(4, 1) = 2^{16} - 3 = 65533$, $A(4, 2) = 2^{65536} - 3$. The second is a number with about 19,729 digits. $A(5, 1)$ is 3 less than a tower of exponents made of 65536 twos; it is a number of unimaginable size.

The Ackermann function has more than one definition. With the definition above, $A(3, 4) = 125$. In the definition used in Horowitz's book, which comes from Tarjan's work, one starts with $A(1, j) = 2^j$ and gets $A(3, 1) = 16$. Whichever definition is chosen, the conclusion does not change: all of them grow equally fast, and in practice $\alpha \leq 4$.

16.6.3 The inverse Ackermann function

Definition 16.6

The **inverse Ackermann function** is defined for $m \geq n \geq 1$ as

$$\alpha(m, n) = \min\{ i \geq 1 : A(i, \lfloor m/n \rfloor) \geq \lg n \}$$

That is, α is the smallest first argument for which the Ackermann function exceeds $\lg n$.

Because A grows so fast, the i needed to push it above a given value is very small. So α grows very slowly. Let us do the computation. Since $m \geq n$, $\lfloor m/n \rfloor \geq 1$ and $A(i, \lfloor m/n \rfloor) \geq A(i, 1)$. From the table, $A(1, 1) = 3$, $A(2, 1) = 5$, $A(3, 1) = 13$ and $A(4, 1) = 65533$. Therefore:

- For $n \leq 2^3 = 8$, $\lg n \leq 3$ and $\alpha \leq 1$.
- For $n \leq 2^5 = 32$, $\alpha \leq 2$.
- For $n \leq 2^{13} = 8192$, $\alpha \leq 3$.
- For $n \leq 2^{65533}$, $\alpha \leq 4$.

The number of atoms in the observable universe is estimated at about 10^{80}, roughly 2^{266}. 2^{65533} is unimaginably larger than that. For every n that can occur on computers, $\alpha \le 4$. Values computed from the definition with a Python script confirmed these thresholds: 1 at $n = 8$, 2 at $n = 9$ and 32, 3 at $n = 33$ and 8192, and 4 at $n = 8193$, 10^6 and 2^{64}.

Theoretically α grows without bound; so $O(\alpha)$ is not constant, and saying "union-find operations are $O(1)$" is not strictly correct. In practice, however, α is a factor of at most 4. The right statement is "almost constant" or "a total cost that is linear in practice".

16.6.4 Recursive Ackermann and the call stack

The Ackermann function is a favorite exam question for showing the power of recursion and the limits of the call stack. The question is: write the function recursively; then show the call stack of the call `A(1, 2)` step by step. The C version is a direct translation of the definition:

```
unsigned long ackermann(unsigned long m,
                        unsigned long n) {
  if (m == 0) return n + 1;
  if (n == 0) return ackermann(m - 1, 1);
  return ackermann(m - 1, ackermann(m, n - 1));
}
```

The three cases of the definition correspond to three lines. Since the results grow quickly, `unsigned long` was chosen as the type; no negative value ever arises. On Windows `unsigned long` is 32 bits, on Linux 64 bits. $A(4, 1) = 65533$ fits in both types; $A(4, 2)$, however, fits in no integer type.

The expansion of `A(1, 2)` is as follows:

$$A(1,2) = A(0, A(1,1)) = A(0, A(0, A(1,0))) = A(0, A(0, A(0,1))) = A(0, A(0, 2)) = A(0, 3) = 4$$

Step	Active calls	Event
1	A(1,2)	Third case: the inner call A(1,1) is needed first.
2	A(1,2) A(1,1)	Third case again: A(1,0) first.
3	A(1,2) A(1,1) A(1,0)	$n = 0$: A(0,1) is called.
4	A(1,2) A(1,1) A(1,0) A(0,1)	$m = 0$: returns 2. A(1,0) = 2.
5	A(1,2) A(1,1) A(0,2)	The inner call gave 2, the outer call is A(0,2): returns 3. A(1,1) = 3.
6	A(1,2) A(0,3)	The inner call gave 3, the outer call is A(0,3): returns 4.
7	—	A(1,2) = 4.

Table 16.11: Trace of `ackermann(1, 2)`. The column "Active calls" shows the call stack from left to right (outermost to innermost).

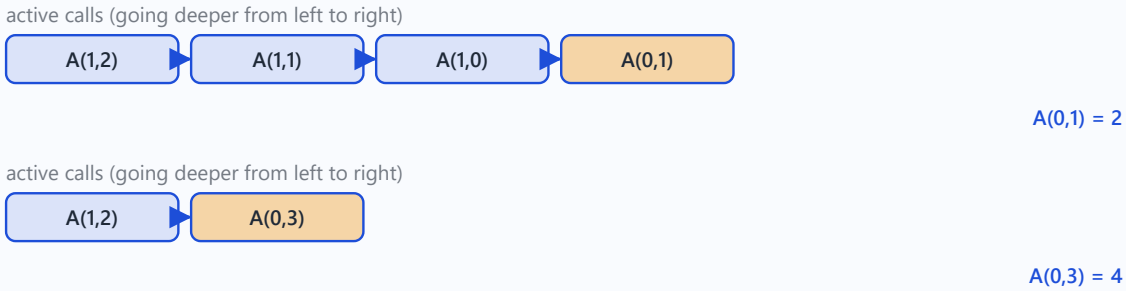


Figure 16.11: Two moments of the trace of `ackermann(1, 2)`. Top: the stack at its deepest: four active calls, the innermost A(0,1) returning 2. Bottom: the last outer call A(0,3) returning 4.

A measurement with an instrumented version showed that `A(1, 2)` makes 6 calls in total and that the stack grows to at most 4 frames. This agrees with the six distinct calls in the table (A(1,2), A(1,1), A(1,0), A(0,1), A(0,2), A(0,3)). The numbers grow quickly: `A(2, 2)` takes 27 calls and depth 8, `A(3, 3)` takes 2,432 calls and depth 63, and `A(4, 0)` takes 107 calls and depth 16. By a calculation in Python with the same counting rule, `A(4, 1)` takes 2,862,984,010 calls and the stack reaches a depth of 65,536 frames. Such a depth can easily exceed the default call stack. `A(4, 2)`, on the other hand, finishes on no computer; even the digits of the result do not fit in memory.

16.6.5 Comparison by measurement

Three versions were compared on the same random sequence of operations. In each operation the roots of two random items were found; in half of the operations, if the roots differed, the two sets were united. Table 16.12 gives the results.

Version	*n*	Hops per find	Deepest level	Time
`union1` + `find1`	100.000	1.108,25	10.130	≈ 2,7–3,1 s
`union2` + `find1`	100.000	0,90	8	≈ 0,007 s
`union2` + `find2`	100.000	0,61	4	≈ 0,006 s
`union2` + `find1`	1.000.000	0,91	10	≈ 0,13 s
`union2` + `find2`	1.000.000	0,61	5	≈ 0,11 s

Table 16.12: *n* items, *2n* random operations (*4n* finds). “Hops” counts only the steps made toward the root. Times vary by machine.

On random operations even the first version does not reach the *n* levels of a chain, but it exceeds a thousand hops per find. The weighting rule reduces this by more than a factor of a thousand. Since weighted trees are already shallow on random input, the contribution of path compression is small here: hops drop from 0.90 to 0.61. When the linking rounds are counted too, the total work increases slightly (about 360 thousand against 492 thousand steps on 100,000 items), while the time stays about the same. The real value of path compression is seen in bad cases where the same deep paths are walked again and again. The measurement on the tree built from equal pairs showed this: 10.00 hops per find dropped to 1.78.

16.6.6 Where do these results come from?

Tarjan is a famous computer scientist who is the author of many fundamental results on graph algorithms. This theorem comes from his joint work with J. van Leeuwen (1984). Results of this kind are the product of research. In undergraduate education one studies for four years and gains a profession. In a doctorate, one often works for four years on a single problem and looks for an answer to a question that has not yet been solved anywhere in the world. Such work is carried out not alone but within a group; a person working in a group keeps learning.

One should not expect these topics to be finished in a week; if one does not work at all, they never get finished. Studying two or three hours regularly every day and reviewing the topic of the week within the same week keeps the topics from piling up. A short but deep structure like union-find is best learned during this review by tracing small examples by hand.

16.7 Applications

16.7.1 Counting connected components

The largest groups of vertices that can reach one another by some path in an undirected graph are called **connected components**. Components partition the vertex set into disjoint parts; so this is a natural application of union-find. At the start every vertex is a separate component. Every edge unites the components in which its two endpoints lie. When all edges have been processed, the number of remaining sets is the number of components.

```
int components(int n, int m, int u[], int v[]) {
  for (int i = 0; i < n; i++) parent[i] = -1;
  int count = n;               /* every vertex separate */
  for (int e = 0; e < m; e++) {
    int a = find2(u[e]), b = find2(v[e]);
    if (a != b) {              /* different component */
      union2(a, b);
      count--;
    }
  }
  return count;
}
```

The routine takes a graph with n vertices and m edges as two arrays: the e-th edge connects the vertices `u[e]` and `v[e]`. The first line builds the parent array from scratch; every vertex is a singleton set. The variable `count` holds the number of components and starts at n. For each edge the loop finds the roots of the two endpoints. If the roots differ, the edge unites two separate components: the roots are united with `union2` and the component count decreases by one. If the roots are the same, the two endpoints of the edge are already in the same component; the edge closes a cycle and the count does not change.

The example graph has 6 vertices and 4 edges: (0, 1), (2, 3), (1, 4) and (3, 2). The last edge is a second edge between 2 and 3. Table 16.13 gives the trace.

Edge	`a`	`b`	Decision	`parent` afterwards	`count`
(0, 1)	0	1	Different: `union2(0, 1)`	`[-2, 0, -1, -1, -1, -1]`	5
(2, 3)	2	3	Different: `union2(2, 3)`	`[-2, 0, -2, 2, -1, -1]`	4
(1, 4)	0	4	Different: `union2(0, 4)`	`[-3, 0, -2, 2, 0, -1]`	3
(3, 2)	2	2	Same: closes a cycle	unchanged	3

Table 16.13: Trace of `components(6, 4, u, v)`. Initially `parent` is all −1 and `count = 6`.

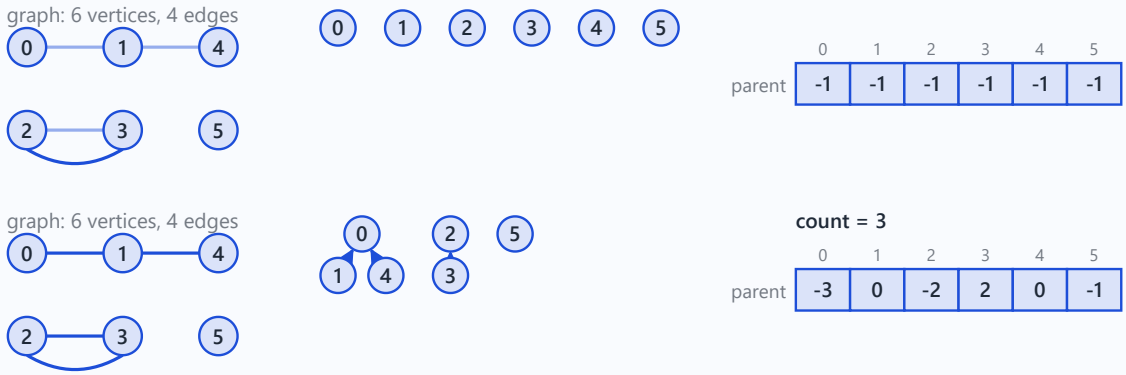


Figure 16.12: Counting components. Graph on the left, forest in the middle, parent array on the right. Top: the start, six singleton sets. Bottom: the result, three components, {0, 1, 4}, {2, 3} and {5}.

The third edge has an interesting moment. The edge is (1, 4), but 1 is not a root. `find2(1)` returns 0, the root of 1, and the union is made between 0 and 4. The tree of 0 has two nodes and that of 4 has one; since *−2 > −1* is false, 4 goes under 0. Both endpoints of the fourth edge reach the root 2, and the edge is skipped. The result is 3: `{0, 1, 4}`, `{2, 3}`, `{5}`. The routine was compiled with the code in the slides and run; it returned 3 and the final array was `[-3, 0, -2, 2, 0, -1]`.

Cost: the first loop is $\Theta(n)$. The edge loop runs m times; each round makes two finds and at most one union. By Theorem 16.3 the total is $O(n + m \cdot \alpha(m + n, n))$; in practice this is linear. Memory is the parent array of n integers. The adjacency list of the graph is not built; edges can be processed as they are read. This property makes union-find attractive compared with depth-first search for very large graphs whose edges arrive in sequence from a file or the network.

16.7.2 Equivalence classes

A relation is an equivalence relation if it is reflexive ($x \equiv x$), symmetric (if $x \equiv y$ then $y \equiv x$) and transitive (if $x \equiv y$ and $y \equiv z$ then $x \equiv z$). An equivalence relation partitions the items into disjoint sets called **equivalence classes**. The classic problem in Horowitz's book is this: a series of declarations of

the form $i \equiv j$ is given; find the classes. For example, the `EQUIVALENCE` declarations that tell a compiler which variables share the same memory area are processed in this way.

Because of transitivity, declarations have indirect consequences: if $7 \equiv 4$ and $0 \equiv 4$, then $7 \equiv 0$, even if this is not declared separately. Union-find gathers these indirect consequences automatically. Each declaration unites the sets of two items:

```
void equivalence(int n, int m, int x[], int y[]) {
  for (int i = 0; i < n; i++) parent[i] = -1;
  for (int k = 0; k < m; k++)
    unite(x[k], y[k]);          /* declaration x[k] ≡ y[k] */
  for (int r = 0; r < n; r++) {     /* every root is a class */
    if (parent[r] >= 0) continue;
    printf("{");
    for (int i = 0; i < n; i++)
      if (find2(i) == r) printf(" %d", i);
    printf(" }\n");
  }
}
```

Let us try it with 12 items and nine declarations: $0 \equiv 4$, $3 \equiv 1$, $6 \equiv 10$, $8 \equiv 9$, $7 \equiv 4$, $6 \equiv 8$, $3 \equiv 5$, $2 \equiv 11$, $11 \equiv 0$. Table 16.14 gives the roots found at each declaration and the classes after the declaration.

Declaration	Roots	Classes afterwards (those with more than one element)
$0 \equiv 4$	0, 4	{0, 4}
$3 \equiv 1$	3, 1	{0, 4}, {1, 3}
$6 \equiv 10$	6, 10	{0, 4}, {1, 3}, {6, 10}
$8 \equiv 9$	8, 9	{0, 4}, {1, 3}, {6, 10}, {8, 9}
$7 \equiv 4$	7, 0	{0, 4, 7}, {1, 3}, {6, 10}, {8, 9}
$6 \equiv 8$	6, 8	{0, 4, 7}, {1, 3}, {6, 8, 9, 10}
$3 \equiv 5$	3, 5	{0, 4, 7}, {1, 3, 5}, {6, 8, 9, 10}
$2 \equiv 11$	2, 11	{0, 4, 7}, {1, 3, 5}, {6, 8, 9, 10}, {2, 11}
$11 \equiv 0$	2, 0	{0, 2, 4, 7, 11}, {1, 3, 5}, {6, 8, 9, 10}

Table 16.14: Processing the equivalence declarations. The “Roots” column is the result of the two `find2` calls inside `unite`.

At the fifth declaration, 4 is no longer a root; its root is 0. At the last declaration, the root of 11 is 2. Since the tree of 2 has two nodes and the tree of 0 has three, $-2 > -3$ is true and 2 goes under 0. The program prints three classes: `{ 0 2 4 7 11 }`, `{ 1 3 5 }` and `{ 6 8 9 10 }`. This output was obtained with the C driver.

The final loop that prints the classes scans all the items for each root and calls `find2` for each item. If the number of classes is k, this part makes $O(k \cdot n)$ finds; if all items are in separate classes, it climbs to $O(n^2)$. A better way is to scan the items once and append each to the list of its root: $O(n \cdot \alpha)$. The simple way was chosen to keep the example short. Processing the declarations is $O(n + m \cdot \alpha)$.

16.7.3 Kruskal's algorithm

Among the trees that span all the vertices of a connected weighted graph, the one with the smallest sum of edge weights is called the **minimum spanning tree**. Kruskal's algorithm builds this tree greedily. The edges are sorted by weight from smallest to largest and examined in order. If `find2` gives different roots for the endpoints of an edge, the edge is taken into the tree and the two sets are united. If it gives the same root, the edge closes a cycle and is rejected.

The reason for the rejection rule is this. If the two endpoints are already in the same set, there is a path between them made of previously chosen edges. The new edge would close this path into a cycle. Since the tree must not have a cycle, the edge cannot be taken. Union-find answers the question "is there a path of chosen edges between these two endpoints?" in nearly constant time. There is no need to search for a path in the graph.

```
typedef struct { int u, v, w; } edge;   /* endpoints and weight */

static int edge_cmp(const void *a, const void *b) {
  const edge *x = a, *y = b;
  return (x->w > y->w) - (x->w < y->w);  /* smaller weight first */
}

int kruskal(int n, int m, edge e[], edge chosen[]) {
  qsort(e, m, sizeof e[0], edge_cmp);
  for (int i = 0; i < n; i++) parent[i] = -1;
  int k = 0;                             /* number of chosen edges */
  for (int t = 0; t < m && k < n - 1; t++) {
    int a = find2(e[t].u), b = find2(e[t].v);
    if (a != b) {                        /* does not close a cycle */
      union2(a, b);
      chosen[k++] = e[t];
    }                                    /* a == b: edge is rejected */
  }
  return k;                              /* if n-1, the graph is connected */
}
```

An edge is kept as a structure: two endpoints and a weight. `qsort` is the sorting routine of the standard library; the comparison function looks at the weights of the two edges. The expression `(x->w > y->w) - (x->w < y->w)` gives −1, 0 or 1. Writing `x->w - y->w` would be shorter, but it can cause integer overflow with very large weights. The loop ends early when $n - 1$ edges have been

chosen, because a tree on n vertices has exactly $n - 1$ edges. The routine returns the number of chosen edges. If this number is less than $n - 1$, the graph is not connected and the result is a spanning forest, not a spanning tree.

Let the edges of an example graph with 5 vertices be as follows, written as (endpoint, endpoint, weight): (0, 1, 4), (0, 2, 1), (1, 2, 2), (1, 3, 5), (2, 3, 8), (3, 4, 3), (2, 4, 9). After sorting, the weights come in the order 1, 2, 3, 4, 5, 8, 9.

Edge	Roots	Decision	`parent` afterwards
(0, 2, 1)	0, 2	Taken	`[-2, -1, 0, -1, -1]`
(1, 2, 2)	1, 0	Taken	`[-3, 0, 0, -1, -1]`
(3, 4, 3)	3, 4	Taken	`[-3, 0, 0, -2, 3]`
(0, 1, 4)	0, 0	Rejected: cycle through the path 0–2–1	unchanged
(1, 3, 5)	0, 3	Taken	`[-5, 0, 0, 0, 3]`

Table 16.15: Kruskal trace. The loop ends when four edges have been chosen ($n - 1 = 4$); (2, 3, 8) and (2, 4, 9) are never examined.

The chosen edges are (0, 2), (1, 2), (3, 4) and (1, 3); the total weight is $1 + 2 + 3 + 5 = 11$. The edge (0, 1, 4) is rejected, because the path 0–2–1 between 0 and 1 has already been chosen. The program gave the same four edges and the total 11.

The cost has two parts. Sorting is $O(m \lg m)$. The union-find part is at most $2m$ finds and $n - 1$ unions: $O(m \cdot \alpha)$. Sorting determines the total: $O(m \lg m)$. Without the nearly constant cost of union-find, for example if the first version were used, the loop part would climb to $O(m \cdot n)$ and exceed the sorting.

16.7.4 Other areas of use

- **Network connectivity.** Answering questions like “are these two computers connected?” online, as connections are added. Sedgewick's book introduces union-find with this example.
- **Image processing.** Splitting neighboring pixels of the same color into regions. This operation is called connected component labeling; each pixel is an item and each adjacency is a union.
- **Percolation.** Testing whether an open path from top to bottom exists in a grid made of randomly opened cells. The core of Monte Carlo experiments is this question asked after each newly opened cell.
- **Compilers.** Unification, which equates two type expressions in type inference, and alias analysis, which finds which pointers may point to the same memory.
- **Offline lowest common ancestor.** Tarjan's algorithm finds the lowest common ancestor (LCA) of node pairs known in advance in a tree, in a single traversal with union-find.

In all these applications the problem is different, but the underlying structure is the same: groups only grow and never split, and one often asks “are they in the same group?” If items must be removed from groups or a group must be split in two, union-find is not suitable; it does not support these

operations.

16.8 Exam question: the contents of the parent array

Questions of this kind measure not memorizing code but being able to trace it by hand. The question is as follows:

Example 16.2

`union2` and `find2` are given. The sets are $S_1 = \{0, 6, 7, 8\}$, $S_2 = \{1, 4, 9\}$ and $S_3 = \{2, 3, 5\}$ (Figure 16.13). (i) What are the initial contents of the parent array? (ii) What is the parent array after the calls `union2(4, 2); find2(5); union2(0, 4); find2(7); find2(3);` ? If (i) is wrong, (ii) is not graded.

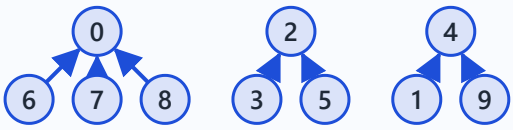


Figure 16.13: The three sets in the exam question. The roots are 0, 2 and 4.

(i) Since `union2` is used, the roots hold minus the node count, not −1. This is the trap of the question: a student who writes −1 at the roots in (i) also gets (ii) wrong, because the weight comparisons are made with wrong numbers. S_1 has 4 items, S_2 has 3 and S_3 has 3:

parent = [−4, 4, −3, 2, −3, 2, 0, 0, 0, 4]

(ii) The calls are traced in order.

Call	Reason	`parent` afterwards
`union2(4, 2)`	*−3 > −3* is false (equal) → 2 is attached to 4; `parent[4] = -6`	`[-4, 4, 4, 2, -6, 2, 0, 0, 0, 4]`
`find2(5)`	Path 5 → 2 → 4; 5 is attached directly to 4 (2 is already attached)	`[-4, 4, 4, 2, -6, 4, 0, 0, 0, 4]`
`union2(0, 4)`	*−4 > −6* is true → 0 is attached to 4; `parent[4] = -10`	`[4, 4, 4, 2, -10, 4, 0, 0, 0, 4]`
`find2(7)`	Path 7 → 0 → 4; 7 is attached to the root	`[4, 4, 4, 2, -10, 4, 0, 4, 0, 4]`
`find2(3)`	Path 3 → 2 → 4; 3 is attached to the root	`[4, 4, 4, 4, -10, 4, 0, 4, 0, 4]`

Table 16.16: Part (ii) of the exam question, step by step.

The result is `[4, 4, 4, 4, -10, 4, 0, 4, 0, 4]`. All ten items are gathered in one set, with root 4. Only 6 and 8 still reach the root through 0; find has not yet been called for them. All the intermediate arrays were obtained exactly in the driver compiled with the code in the slides.

The note "2 is already attached" in the second step is the extra round of `find2`. The path is 5 → 2 → 4; the `while` loop first attaches 5 and then 2 to the root. Since 2 is already attached to 4, the second assignment does not change the value.

16.9 Complexity summary

Version	Union	Find (worst case)	*n* − 1 unions + *m* finds
`union1` + `find1`	O(1)	O(n)	O(n + m · n)
`union2` (weighted)	O(1)	O(lg n)	O(n + m lg n)
`union2` + `find2`	O(1)	O(lg n) per operation	O(n + m · α(m + n, n))

Table 16.17: Complexity of the union-find versions. *n* is the number of items and *m* the number of finds.

Memory is the same in every version: just the parent array of *n* integers. Since the weighting rule stores the node count in the negative value at the root, it needs no extra array. Because $\alpha \le 4$, the last row is linear in practice.

The essence of the table can be said in a single sentence: union by itself is always *O(1)*; what determines the cost is the depth of the tree, that is, find. The weighting rule bounds the depth and the collapsing rule shortens the paths. That union is constant is misleading, because `union` requires roots, and finding a root is a call to `find2`. In practice a request to "unite two items" therefore costs two finds plus one union.

16.10 Common mistakes

1. **Giving union an arbitrary node instead of a root.** In `union1(i, j)`, if *i* is not a root, the assignment `parent[i] = j` cuts the link between *i* and its old parent. *i* and its subtree move to the set of *j*, while the rest of the old tree stays in the old set. Two sets have not been merged; one has been split. In the example forest, `union1(9, 2)` moves only 9; 1 and 4 stay in the old set. The fix is to find the roots first with `find`.
2. **Uniting two roots in the same set, that is, a root with itself** (*i == j*). `union1(r, r)` links the root to itself and later finds fall into an infinite loop. `union2(r, r)` doubles the node count. The roots must be compared before uniting.
3. **Reading the comparison of negative values backward in the weighting rule.** *−2 > −4* means "fewer nodes". A reversed comparison does not break the sets, but it turns the trees into chains, and the error is noticed only by measurement.

4. **Forgetting to update the node count at the root (`p`) in the union.** The old count stays at the new root and later comparisons are made with wrong weights. Computing `p` before either of the cells is changed is equally important.
5. **Attaching in `find2` without saving the parent into `s`.** The rest of the path is lost; the path stays uncompressed.
6. **Not filling the parent array with −1 at the start.** Global arrays start at zero; the contents of an array obtained with `malloc` are indeterminate. In an array full of zeros the parent of every node is 0 and the parent of 0 is also 0: `find` falls into an infinite loop at 0.
7. **Not checking the return of `malloc` and forgetting `free`.** In the dynamic version, if the `parent` array is not given back, the program leaks *4n* bytes on every call. Before giving the structure back, the inner array must be given back.

16.11 Exercises and solutions

Exercise 1. *n = 6*, `parent` all −1. What is `parent` after the calls `union2(0, 1); union2(2, 3); union2(0, 2); union2(4, 5);` ?

Solution. The first two calls unite equal trees: `parent[1] = 0`, `parent[0] = -2`; `parent[3] = 2`, `parent[2] = -2`. In the third call both roots hold −2; in a tie, 2 goes under 0 and `parent[0] = -4`. The fourth call writes `parent[5] = 4`, `parent[4] = -2`. The result is `[-4, 0, 0, 2, -2, 4]` : root 0 has 4 nodes and root 4 has 2 nodes. Verified with the driver.

Exercise 2. With the weighting rule, what is the maximum depth of a tree with 16 nodes?

Solution. By Theorem 16.2, *⌊lg 16⌋ + 1 = 5* levels. The tree that reaches the bound is built from equal pairs: 8 pairs, 4 fours, 2 eights, 1 sixteen. The measurement also gave 5 levels.

Exercise 3. Write a union rule that keeps height instead of weight. What changes?

Solution. −(number of levels) is kept at the root. The lower tree is attached under the higher one; if the heights are equal, one goes under the other and the number of levels increases by 1. This method is called **union by rank**:

```
void union3(int i, int j) {
  /* height rule; i and j are roots; at a root -(number of levels) */
  if (parent[i] > parent[j]) {        /* the tree of i is lower */
    parent[i] = j;
  } else if (parent[i] < parent[j]) { /* the tree of j is lower */
    parent[j] = i;
  } else {                      /* equal heights */
    parent[j] = i;
    parent[i]--;                   /* number of levels grows by 1 */
  }
}
```

The comparison is again between negative values: the larger value points to the lower tree. When two trees of different heights are united, the height does not change, because the lower tree goes under the root of the higher one and reaches at most its height. With the sequence of operations of Exercise 1 the result is `[-3, 0, 0, 2, -2, 4]`: the same trees, but root 0 holds the number of levels, 3, instead of the node count, 4. With equal pairs of 16 nodes, 5 levels again result. The depth is again bounded by $O(\lg n)$; the proof is the same as for the weighting rule (a tree with d levels has at least 2^{d-1} nodes). When used together with the collapsing rule, path compression lowers the height but the root does not know this; the stored value is no longer the true height but an upper bound on it. That is why it is called “rank” rather than height.

Exercise 4. Write `find2` recursively.

Solution. One-line path compression: `int find(int i) { return parent[i] < 0 ? i : (parent[i] = find(parent[i])); }`. A detailed explanation and a memory comparison are in 16.5.5.

Exercise 5. Why does the rejection of an edge in Kruskal's algorithm mean a cycle?

Solution. An edge is rejected if its endpoints are in the same set. Being in the same set means there is a path of previously chosen edges between the two endpoints, because sets are merged only by chosen edges. The new edge connects the two ends of this path directly and closes the path into a cycle.

Exercise 6. What is the cost of finding the number of components of a graph with n vertices and m edges?

Solution. Building the parent array is $O(n)$; the edge loop is $2m$ finds and at most $n - 1$ unions. With the two rules together the total is $O(n + m \cdot \alpha(m + n, n))$; in practice it is linear.

16.12 Summary

- **Model.** Disjoint sets are represented by trees whose root is the set name and whose links go from child to parent. There are two operations: union and find. Two items are in the same set if their roots are the same.
- **Representation.** The whole forest sits in a single `int` array: `parent[i]` is the parent of i; a negative value is found at a root. If the array is built with `malloc`, `NULL` is checked after every allocation and the blocks are given back in reverse order.
- **First version.** `union1` is a single assignment; `find1` walks to the root and takes as long as the depth of the node.
- **Worst case.** Arbitrary linking without a rule can produce a chain: after the sequence `union1(i, i+1)`, n finds take exactly $n(n - 1)/2$ hops, that is, $\Theta(n^2)$. Random tests easily miss this input.
- **Weighting rule.** The small tree is attached under the large one. The node count is stored at the root with a minus sign; no extra array is needed. The depth does not exceed $\lfloor \lg m \rfloor + 1$, and the bound is reached exactly by equal pairs.

- **Collapsing rule.** During find, every node on the path is attached directly to the root. The first call is about twice as expensive, but later searches finish in one stroke. The old parent must be saved before linking.
- **Result.** The two rules together give a cost per operation proportional to α (Tarjan–van Leeuwen). The inverse Ackermann function does not exceed 4 for $n \le 2^{65533}$; the cost is almost *O(1)* in practice.
- **Applications.** Counting components, equivalence classes and Kruskal's algorithm use the same pattern: find the roots of the two endpoints; if they differ, unite them; if they are the same, there is a cycle.

16.13 Key terms

disjoint sets union-find union find forest root parent parent array weighting rule union by rank collapsing rule path compression worst case amortized analysis Ackermann function inverse Ackermann function call stack stack overflow connected component minimum spanning tree memory leak dangling pointer pointer to pointer

---

16.14 Test Yourself

1. How many sets are there in the array `parent = [-1, 0, 0, 1, 1, -1, 5]`? What does `find1(4)` return and how many hops does it make?

   Answer

   There are two cells with negative values (0 and 5); two sets: {0, 1, 2, 3, 4} and {5, 6}. `find1(4)` follows the path 4 → 1 → 0 and returns 0 after 2 hops.

2. The calls `union1(i, i+1)` for *i = 0, ..., 6* are applied to *n = 8* singleton sets. Then one `find1` is called for each item. How many hops are made in total?

   Answer

   n(n − 1)/2 = 8 · 7 / 2 = 28. `find1(0)` makes 7 hops, `find1(1)` 6, ... and `find1(7)` 0.

3. With `parent = [-3, 0, 0, -5, 3, 3, 3, 3]`, `union2(0, 3)` is called. What does the array become?

   Answer

   p = −8. −3 > −5 is true: the tree of 0 is smaller and goes under 3. The result is `[3, 0, 0, -8, 3, 3, 3, 3]`.

4. With `parent = [-8, 0, 0, 2, 0, 4, 4, 6]`, `find2(3)` is called. How many rounds does each loop run, and what does the array become?

   Answer

   The path is 3 → 2 → 0. The `for` loop makes 2 hops and the `while` loop runs 2 rounds: 3 is attached to the root and an extra round is made for 2. The result is `[-8, 0, 0, 0, 0, 4, 4, 6]`, and the return value is 0.

5. In a tree with 1024 nodes built with the weighting rule, how many hops does a single `find1` make at most?

   Answer

   The number of levels is at most $\lfloor \lg 1024 \rfloor + 1 = 11$. From the deepest node to the root there are 10 edges: at most 10 hops.

6. What is $\alpha(n, n)$ for $n = 1000$? (With the A of the definition.)

   Answer

   $\lfloor n/n \rfloor = 1$, $\lg 1000 \approx 9.97$. $A(1, 1) = 3$ and $A(2, 1) = 5$ are not enough; $A(3, 1) = 13 \geq 9.97$. The answer is 3.

7. What does `ackermann(2, 1)` return, and how many calls are made in total?

   Answer

   $A(2, 1) = 2 \cdot 1 + 3 = 5$. In an instrumented run, 14 calls were made.

8. The edges of a graph with 5 vertices are (0, 1), (1, 2), (2, 0), (3, 4). What does `components` return? Which edge does not change the count?

   Answer

   It returns 2: {0, 1, 2} and {3, 4}. Both endpoints of the edge (2, 0) reach the root 0; the edge closes the cycle 0–1–2 and the count does not change.

9. What would be lost if `sets_free` took the parameter as `sets *k` instead of `sets **sp`?

Answer

The routine could still give the blocks back, but it could not set the caller's pointer to `NULL`. The caller's variable would keep pointing at memory that had been given back, that is, it would be a dangling pointer. A second delete call would try to give the same block back twice, and undefined behavior would result.

16.15 Further reading

- E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2nd ed., Section 5.10 (Set Representation). The parent array, the weighting rule and the collapsing rule are presented in the same order as in this chapter, with their lemmas.
- T. H. Cormen et al., *Introduction to Algorithms*, the chapter "Data Structures for Disjoint Sets". It gives a detailed proof of the bound with α for union by rank and path compression.
- R. Sedgewick, *Algorithms in C*, Chapter 1. It develops union-find step by step with the network connectivity example and compares the versions with measurements.
- R. E. Tarjan, J. van Leeuwen, "Worst-case analysis of set union algorithms", *Journal of the ACM* 31(2), 1984. It is the source of Theorem 16.3; it studies many variants of path compression together.

16.16 References

1. E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, Section 5.10 (Set Representation).
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms* (CLRS), the chapter "Data Structures for Disjoint Sets".
3. R. Sedgewick, *Algorithms in C*, Chapter 1 (connectivity with union-find).
4. R. E. Tarjan, J. van Leeuwen, "Worst-case analysis of set union algorithms", *J. ACM* 31(2), 1984.

PART IV · CHAPTER 17

Minimum Spanning Trees

A subgraph that connects all the vertices of a graph without forming a cycle is called a spanning tree. If the edges have a cost, we look among these trees for the one with the smallest total cost. Connecting villages to the electric grid with the least cable, or adding computers to a network with the cheapest links, are examples of this problem. The chapter starts with the definition, the number and the properties of spanning trees. Then come two classic greedy algorithms. Kruskal's algorithm tries edges from cheapest to most expensive and catches cycles with sets (union-find). Prim's algorithm grows a single tree by the cheapest edge and is coded with arrays or with a heap. The C code of both algorithms is explained line by line, traced step by step on the same example graph, and its cost is computed with justification. The main lesson of the chapter is this: the same algorithm runs at a completely different speed depending on the data structure chosen.

LEARNING OUTCOMES

Define
Explain the spanning tree, tree edges and nontree edges, and the spanning trees produced by depth-first or breadth-first search.

Count
Discuss ways of generating the spanning trees of a graph, the notion of isomorphism and why the number of trees grows so fast.

Kruskal
Code in C and trace by hand Kruskal's algorithm, which selects edges by increasing cost and catches cycles with the union-find structure.

Prim
Implement Prim's algorithm, which grows a single tree, with $O(n^2)$ arrays and with a heap.

Analyze
Compare the bounds $O(e \log e)$, $O(n^2)$ and $O(e \log n)$; choose the algorithm according to whether the graph is dense or sparse.

Approach
See how a min-heap and disjoint sets speed up an algorithm: first discover the data structure, then code.

17.1 Spanning trees

A graph $G = (V, E)$ consists of vertices (V) and edges (E) that join pairs of vertices. In this chapter the number of vertices is written $n = |V|$ and the number of edges $e = |E|$, and unless stated otherwise the graph is undirected. If there is a path between every pair of vertices, the graph is **connected**. A path that returns to the vertex it started from is a **cycle**. A connected graph without cycles is a **tree**; there is no need to choose a root in a tree.

The whole question of this chapter fits in one sentence: which of a graph's edges are enough to connect its vertices, and which of these is the cheapest? Let us first make the word “enough” precise.

17.1.1 A traversal splits the edges into two groups

There are two basic ways to traverse a graph. **Depth-first search** (DFS) moves to a neighbor and descends as deep as possible; when there is nowhere left to go it backtracks. It is written with recursion or with a stack. **Breadth-first search** (BFS) visits all neighbors of the start vertex, then their neighbors; it is written with a queue. Both mark each vertex once and finish in $O(n + e)$ time with adjacency lists.

When a traversal has visited all the vertices of a connected graph, the edges have been divided into two groups:

- **T: tree edges.** These are the edges that the traversal processed, that is, edges that lead to a vertex not seen before. Every vertex other than the start is reached exactly once by such an edge. So the number of tree edges is exactly $n - 1$.
- **N: nontree edges.** These are the edges the traversal looked at but did not process; their other endpoint is already marked. In DFS they are also called back edges, in BFS cross edges.

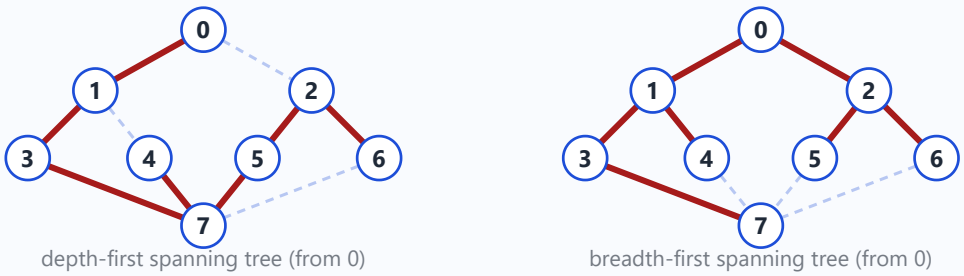


Figure 17.1: The DFS (left) and BFS (right) spanning trees of the eight-vertex example graph, starting from 0. The thick red edges are tree edges; the dashed ones are nontree edges. DFS: N = {(0,2), (1,4), (6,7)}; BFS: N =

{(4,7), (5,7), (6,7)}.

The graph in Figure 17.1 has 10 edges: (0,1), (0,2), (1,3), (1,4), (2,5), (2,6), (3,7), (4,7), (5,7), (6,7). The adjacency lists are sorted in increasing order. Table 17.1 shows in order which edge DFS processes or skips, and why.

Order	Current vertex	Edge examined	Other endpoint	Result
1	0	(0,1)	1 new	tree
2	1	(1,0)	0 is marked	edge we came by
3	1	(1,3)	3 new	tree
4	3	(3,7)	7 new	tree
5	7	(7,4)	4 new	tree
6	4	(4,1)	1 is marked	nontree: (1,4)
7	7	(7,5)	5 new	tree
8	5	(5,2)	2 new	tree
9	2	(2,0)	0 is marked	nontree: (0,2)
10	2	(2,6)	6 new	tree
11	6	(6,7)	7 is marked	nontree: (6,7)

Table 17.1: DFS from 0 on the graph in Figure 17.1. Each row is a look at one edge; “tree” is a processed edge, “nontree” is an edge skipped because it leads to a marked vertex.

The seven tree edges are (0,1), (1,3), (3,7), (7,4), (7,5), (5,2), (2,6); this agrees with the expectation $n - 1 = 7$. The other edges looked at during the returns (for example 1 in the list of 3) were not written into the table because they give no new information. When BFS starts from 0 on the same graph, it first marks 1 and 2, then 3, 4, 5, 6 and finally 7. The tree edges are (0,1), (0,2), (1,3), (1,4), (2,5), (2,6), (3,7). The first edge to reach 7 is (3,7); (4,7), (5,7), (6,7) remain outside the tree.

The two traversals produced two different trees from the same graph. Both span all the vertices, and neither has a cycle. So the “edges enough to connect all the vertices” are not a single set. Other trees come out again when the start vertex or the order of the neighbors changes.

17.1.2 Definition of a spanning tree

Definition 17.1

Let $G = (V, E)$ be a connected graph. A tree that contains all the vertices of G and whose edges are chosen from E is called a **spanning tree** of G. A spanning tree has n vertices and $n - 1$ edges; it has no cycle.

The two conditions in the definition complement each other. “Contains all the vertices” makes the tree spanning: no vertex is left out. “Its edges are chosen from E” forbids inventing new connections: only edges that exist in the graph can be used. A graph that is not connected has no spanning tree. In such a graph each component has its own spanning tree, and their collection is called a **spanning forest**.

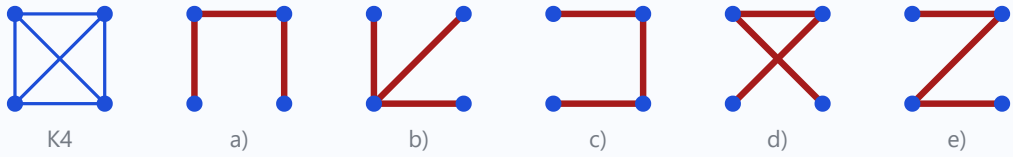


Figure 17.2: The complete graph on four vertices, K_4, and five of its spanning trees. a, c, d, e are paths; b is a star.

The K_4 on the left of Figure 17.2 is the **complete graph**: every pair of vertices is joined by an edge, for a total of $4 \cdot 3 / 2 = 6$ edges. Each of the five drawings next to it selects three of these six edges. All four vertices are used in each and none has a cycle; all five are spanning trees.

Of these five trees, a, c, d and e have the same shape: four vertices lined up on a single line. Only the position of each vertex along the path differs. b is different: one vertex is connected directly to the other three. The first shape is called a **path**, the second a **star**.

Definition 17.2

Two graphs are **isomorphic** if their vertices can be matched one to one and, under this matching, the edges of one correspond exactly to the edges of the other. Isomorphism is denoted by $\equiv$.

For the trees in the figure we write $a \equiv c \equiv d \equiv e$; b is isomorphic to none of them. A quick way to see this is to look at the degrees. In a path two vertices have degree 1 and two have degree 2. In a star one vertex has degree 3 and three have degree 1. Two isomorphic graphs must have the same degree sequence; since it differs here, a and b cannot be isomorphic. The converse is not true: two graphs with the same degree sequence need not be isomorphic. So comparing degrees can only refute isomorphism, not prove it.

Counting the spanning trees of K_4 is easy. Three of the six edges can be chosen in $C(6, 3) = 20$ ways. Four of these choices are triangles: choosing three of the four vertices and taking the three edges among them forms a cycle and leaves the fourth vertex out. All of the remaining 16 choices are trees. The number of stars equals the number of choices of the center: 4. The number of paths is found from the orderings of the four vertices. There are $4! = 24$ orderings, but each path is read twice (from either end). The number of paths is $24 / 2 = 12$. The total is $12 + 4 = 16$.

Theorem 17.1 (Cayley's formula)

The number of spanning trees of the complete graph K_n on n vertices is n^{n-2}.

For $n = 4$ the formula gives $4^2 = 16$, which agrees with the count above. The number grows very fast: K_5 has 125 spanning trees and K_{10} has a hundred million. The proof is outside the scope of this chapter. What matters here is the consequence: generating spanning trees one by one and choosing the best is a hopeless task even for small graphs. Once the edges are given weights, the question "how many?" turns into "which is the cheapest?", and the size of this number confronts us again.

17.1.3 Isomorphism, drawings and planarity

A graph is not just its picture on paper. The graph is determined by its vertices and edges; the positions of vertices on the paper are a choice. The same graph can be drawn in forms that look nothing alike. Figure 17.3 is a striking example.

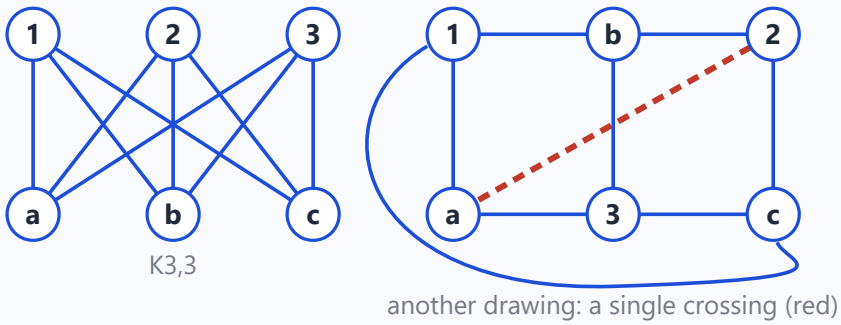


Figure 17.3: Two drawings of the same graph ($K_{3,3}$). On the left the nine edges cross at many points. In the drawing on the right only the red dashed edge (a,2) crosses the edge (b,3).

$K_{3,3}$ consists of six vertices split into two groups: {1, 2, 3} and {a, b, c}. Each vertex of each group is connected to the three vertices of the other group and not to those in its own group. There are $3 \cdot 3 = 9$ edges in total. The drawing on the left applies this definition directly and the edges tangle with one another. In the drawing on the right the vertices are placed elsewhere and the edge (1,c) is routed around the outside. All nine edges are still there, but now only one crossing remains.

Showing that two drawings are the same graph means showing isomorphism: every vertex on the left corresponds to a vertex on the right, and the edges are matched accordingly. The matching here is easy, because the labels have been kept. Given two unlabeled drawings, the problem gets harder. Simple criteria such as the degree sequence are not enough; splitting the vertices into layers with DFS or BFS helps narrow the matching but gives no general solution. Graph isomorphism is a famous problem whose status, whether an efficient algorithm is known, was debated for a long time. The instructor suggests this problem as a nice senior-year project: a program that turns one drawing into

the other with a three-dimensional display and finds which vertices have to move. The instructor then adds that questions of this kind are really hard and can be tackled seriously only at the master's or doctoral level. What is stressed is not that the problem is unsolvable, but that it takes effort.

The question of crossings has a name as well. A graph that can be drawn without any edges crossing is called **planar**. $K_{3,3}$ is not planar: however the vertices are placed, at least one crossing remains. The drawing on the right reaches this lower bound. The question has a concrete counterpart in engineering. On a printed circuit board, conductor tracks cannot cross on the same layer. A circuit that must be connected in the form of $K_{3,3}$ does not fit on a single layer and needs at least two.

17.1.4 How many different spanning trees are there?

The crudest way to generate spanning trees is to try all possible edge sets. Since a spanning tree has exactly $n - 1$ edges, $n - 1$ of the e edges are chosen. This can be done in $C(e, n - 1)$ ways. For each choice it is checked whether there is a cycle; those without cycles are trees. If isomorphic trees are not to be counted separately, the isomorphic ones must also be weeded out.

Consider a graph with four vertices in the shape of a square: the vertices A, B, C, D are connected in order, and there is also a diagonal between A and C. Three of the five edges can be chosen in $C(5, 3) = 10$ ways. Two of these are triangles (ABC and ACD) and form a cycle; 8 spanning trees remain.

A cleverer method is to start from one spanning tree and move to the other trees by exchanging edges. Figure 17.4 shows the method step by step.

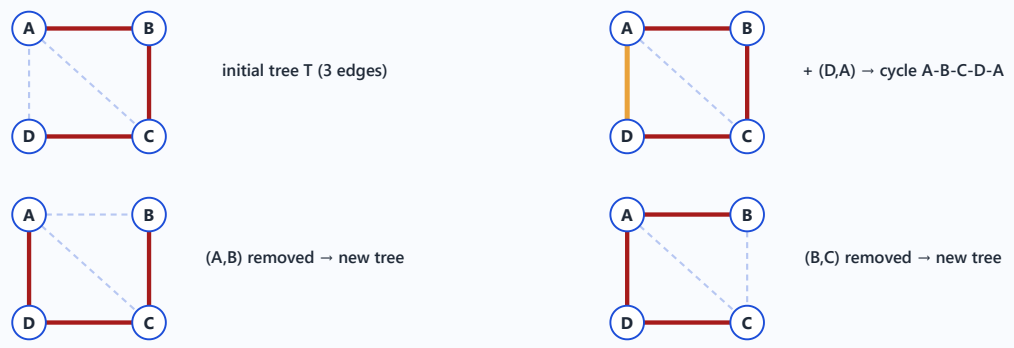


Figure 17.4: Generating spanning trees by exchanging along a cycle. Top left: the tree T found by DFS. Top right: the nontree edge (D,A) has been added and the cycle A-B-C-D-A has formed. Bottom row: removing (A,B) and (B,C) from the cycle gives two new trees.

1. First a spanning tree is found, for example by DFS. In the figure, T = {(A,B), (B,C), (C,D)}.
2. A nontree edge is added. Since there is a single path between every pair of vertices in a tree, the new edge forms exactly one cycle. When (D,A) is added, the cycle is A-B-C-D-A.
3. One of the edges on the cycle is removed. The cycle is broken, the vertices stay connected, and the result is a new spanning tree. Removing (A,B) gives {(B,C), (C,D), (D,A)}. Putting (A,B) back and removing (B,C) gives another one, and removing (C,D) a third.

4. The process is repeated for all nontree edges and for every new tree found. On this graph all 8 spanning trees are reached in the end.

The method works but is long. The same tree is reached many times by different routes, and every new tree must be checked separately for whether it was found before. That is why it is not preferred for counting trees or finding the cheapest tree. When an exact count is needed, Kirchhoff's matrix-tree theorem is used. This theorem gives the number in one stroke by taking the determinant of a matrix built from the degree and adjacency information of the graph. By the theorem, the eight-vertex graph in Figure 17.1 has 64 spanning trees and the seven-village graph, the main example of the chapter, has 46. The details of the theorem are an advanced topic.

17.1.5 Properties of spanning trees

A spanning tree has four properties that are used in algorithms. Figure 17.5 shows them on the BFS spanning tree.

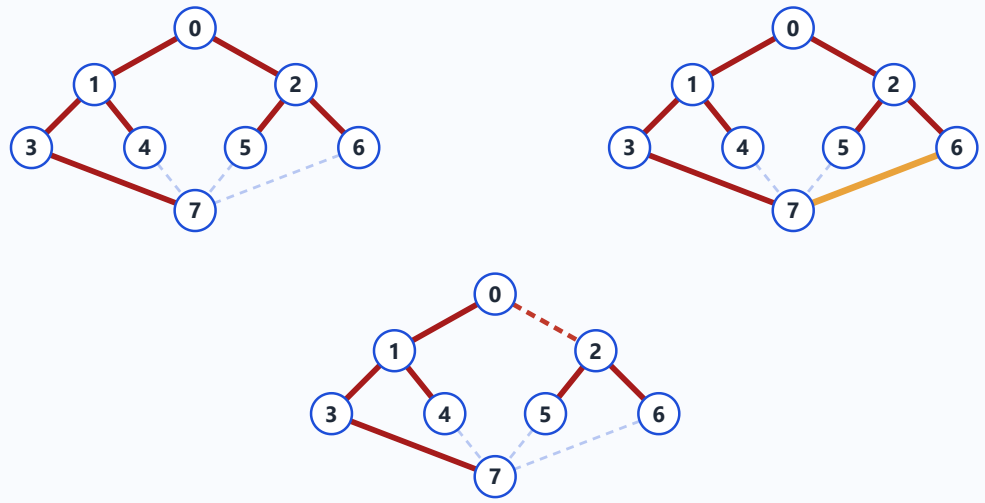


Figure 17.5: Properties of a spanning tree. Left: the BFS spanning tree: 8 vertices, 7 edges. Middle: the nontree edge (6,7) has been added and the cycle 6-2-0-1-3-7-6 has formed. Right: the tree edge (0,2) has been deleted and {2, 5, 6} has been cut off from the tree.

Theorem 17.2 (Properties of a tree)

For a graph T with n vertices, any two of the following three properties imply the third: (i) T is connected; (ii) T has no cycle; (iii) T has exactly $n - 1$ edges. In that case T is a tree and the following also hold:

1. There is a single path between every pair of vertices.
2. Adding an edge to the tree creates exactly one cycle.
3. Deleting any edge of the tree splits it into two pieces.

Proof. From (i) and (ii) to (iii): a connected graph without cycles has at least one leaf (a vertex of degree 1). One end of a longest path must be a leaf; otherwise the path could be extended or a cycle would close. Deleting a leaf and the single edge touching it neither breaks connectivity nor creates a cycle. When this deletion is repeated $n - 1$ times, a single vertex and zero edges remain; so there were $n - 1$ edges at the start. The unique-path property follows from this too: if there were two different paths between two vertices, these paths would join to form a cycle. When an edge (u, v) is added, the single tree path between u and v is closed by the new edge and exactly one cycle forms. When an edge is deleted, the single path between its endpoints is broken and the endpoints end up in separate pieces. The other two directions are shown by similar counting. ■

In the figure these properties can be seen one by one. The BFS tree in the left drawing connects 8 vertices with 7 edges. In the middle drawing, when (6,7) is added, the new edge joins the single tree path 6-2-0-1-3-7 from 6 to 7 and the cycle 6-2-0-1-3-7-6 is closed. This graph is no longer a tree. In the right drawing, when (0,2) is deleted, vertices 2, 5 and 6 can no longer be connected to 0. A spanning tree is the subgraph with the fewest edges among those that keep the vertices connected. One edge too many means a cycle, one edge too few means a disconnection.

Using the second and third properties together gives a method for generating spanning trees: add a nontree edge, and remove an edge from the single cycle that forms. The correctness proofs of the minimum spanning tree algorithms also rest on the same two properties.

The applications of spanning trees also come from these properties:

- **Discarding redundant links.** In a computer network, cycles can make a broadcast packet circulate forever. A spanning tree of the network is chosen and only its links are used.
- **Broadcast tree.** To send a message from one source to all vertices, the edges of the spanning tree are followed; each vertex receives the message once.
- **Electric circuits.** Kirchhoff used spanning trees to find an independent set of circuit equations. Each nontree edge forms a single cycle; each of these cycles gives an independent voltage equation.

17.2 The minimum-cost spanning tree

17.2.1 The weighted graph and the problem

In real problems the edges are not equal. The road between two cities has a length, the line between two computers has a rental fee, the cable to be laid between two villages has a cost. A graph whose edges have been assigned such a number is a **weighted graph**, and $G(V, E, w)$ is written instead of $G(V, E)$. Here $w(u, v)$ is the weight of the edge (u, v). In this chapter the weight is also called the cost.

Definition 17.3

The **cost** of a spanning tree of a connected weighted graph is the sum of the weights of the edges in the tree. The spanning tree with the smallest cost is called the **minimum spanning tree** (MST).

The definition says “the smallest one”, not “the smallest ones”; even so, more than one spanning tree can have the same smallest cost. If all edge weights are distinct, the minimum spanning tree is unique. If there are equal weights there may be several trees, but all have the same cost.

Two example problems make this concrete. In the first, the vertices are the computers of an institution, and the edge weights are the cost of laying a line between two computers. We look for the cheapest set of lines that connects all the computers to one another. In the second, electric or fiber-optic cable is to be taken to seven villages. A feasibility study has worked out how many kilometers of cable are needed from which village to which. For pairs of villages that cannot be connected directly because there is a mountain or a lake between them, the cost is infinite; these edges are not written into the graph at all.

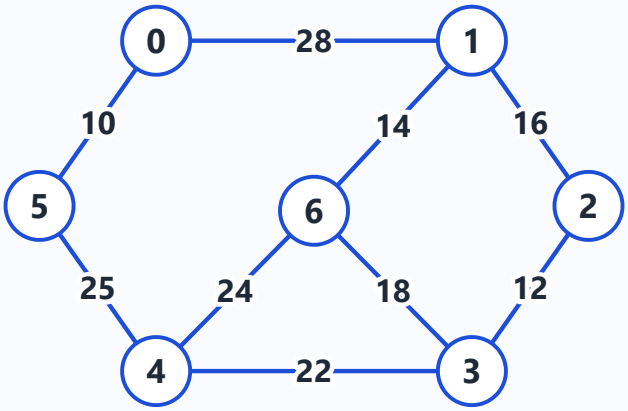


Figure 17.6: The main example of the chapter: seven villages (0–6) and the lengths in kilometers of the cables that can be laid between them. There are nine edges.

The edges of the graph in Figure 17.6 with their weights are: (0,1) = 28, (0,5) = 10, (1,2) = 16, (1,6) = 14, (2,3) = 12, (3,4) = 22, (3,6) = 18, (4,5) = 25, (4,6) = 24. All the Kruskal and Prim traces in the chapter are on this graph. The graph has 46 spanning trees. Six of the nine edges can be chosen in $C(9, 6) = 84$ ways; 38 of these form cycles. The costs of the 46 trees range from 99 to 129. The cost of the cheapest tree is 99 and it is unique; the two cheapest trees after it have cost 101. These numbers were found by trying all the choices one by one on a computer. The goal is to reach the tree of cost 99 directly, without doing this trial.

The instructor underlines an important point here: when looking for a minimum spanning tree, cycles are avoided. A village receiving electricity by two routes, for example laying a backup line so that "if power does not come from Lower Tepecik, let it come from Upper Tepecik", is not the subject of this problem. A cycle means a backup connection, and a backup connection is extra cost. Networks that require reliability are designed with other problems.

17.2.2 The greedy approach

Many algorithms are known for the minimum spanning tree: the algorithms of Kruskal, Prim and Sollin (Borůvka) are the best known; there is also a method called reverse-delete. This chapter concentrates on the first two; the other two are described briefly.

There are a few general methods in algorithm design. **Backtracking** extends a partial solution step by step and, on reaching a dead end, undoes the last decision. Dynamic programming stores the solutions of small subproblems and builds the larger ones from them. The **greedy method**, on the other hand, makes the choice that looks best at the moment at every step and never changes this choice again. The instructor calls this attitude "stubborn": a greedy algorithm does not go back; once it has taken an edge it does not let it go.

Minimum spanning tree algorithms are greedy and obey three rules:

1. An edge that forms a cycle is not taken.
2. Among the edges that can be chosen, the one with the least cost is preferred.
3. In a graph with n vertices exactly $n - 1$ edges are chosen.

The first rule makes the result a tree and the third makes it span all the vertices; together they are conditions (ii) and (iii) of Theorem 17.2. The second rule keeps the cost small. The difference between the algorithms lies in how they define the set of "edges that can be chosen". In Kruskal's this set is all the edges not yet tried. In Prim's it is only the edges that touch the growing tree.

The slides of the chapter open with a saying attributed to Michelangelo: "Beauty is the purgation of superfluities." A minimum spanning tree is exactly this: the plainest network that preserves connectivity, with the excess thrown away.

17.2.3 Why does the greedy choice give the right result?

Greedy algorithms do not find the best result in most problems. Choosing the largest coin each time to make change uses too many coins in some currency systems. So for the minimum spanning tree we must show why the greedy choice is never regretted. The key to this is the cut property.

Definition 17.4

Splitting the vertices of a graph into two nonempty groups, S and $V - S$, is called a **cut**. An edge with one endpoint in S and the other in $V - S$ **crosses the cut**.

Theorem 17.3 (Cut property)

Take any cut in a connected weighted graph. If (u, v) is the edge of least weight among the edges crossing the cut, then there is a minimum spanning tree that contains (u, v). If the weights are distinct, (u, v) is in every minimum spanning tree.

Proof. Let T be a minimum spanning tree that does not contain (u, v). Add (u, v) to T. By Theorem 17.2 exactly one cycle forms. The cycle leaves u (on the side of S), arrives at v (on the other side) and returns to u. To get back to S from the other side, it must cross the cut at least once more; this crossing is made by a tree edge, say (x, y). Since (u, v) is the lightest edge of the cut, $w(u, v) \le w(x, y)$. Delete (x, y): the cycle is broken and we again obtain a spanning tree. The cost of the new tree is not greater than that of T. Since T is minimum, the costs are equal; the new tree is also minimum and contains (u, v). If the weights are distinct, $w(u, v) < w(x, y)$ and the new tree would be cheaper than T. This is a contradiction; so T must already contain (u, v). ■

Kruskal and Prim select the lightest edge of some cut at every step. In Prim's the cut is obvious: S is the vertices of the tree built so far, and the selected edge is the cheapest edge leaving the tree. In Kruskal's the endpoints of the selected edge stand in two separate clusters. If the cluster of one endpoint is taken as S, the edge is the cheapest of the edges crossing this cut. Had there been a cheaper crossing edge, Kruskal would have looked at it earlier and the two clusters would already have merged. So both algorithms select an edge of a minimum spanning tree at every step, and no choice ever has to be undone later.

The cut property has a twin: the **cycle property**. The heaviest edge on a cycle (if the weights are distinct) is in no minimum spanning tree. The reverse-delete method rests on this property.

Common mistake

Thinking “let every vertex choose the cheapest edge touching it, and that's it”. This rule is a special case of the cut property (the cut between a single vertex and the rest), and the selected edges really do belong to the MST. But it is not sufficient. In the seven-village example, (0,5) is chosen for 0 and 5, (1,6) for 1 and 6, (2,3) for 2 and 3, and (3,4) for 4. Three separate clusters result, and the tree is not completed. Sollin's algorithm finishes the job by repeating this rule on clusters in rounds.

17.3 Kruskal's algorithm

17.3.1 The algorithm

Joseph B. Kruskal published the algorithm in 1956. The idea is one sentence: try the edges in order from cheapest to most expensive; take one that does not form a cycle and discard one that does. The correctness of the algorithm has a proof and rests on the cut property. Reading the proof explains better why the algorithm is built this way. The pseudocode is as follows:

```
T = {};                    /* chosen edges */
while (|T| < n-1 && E ≠ ∅) {
    pick the cheapest edge (v,w) from E;
    E = E - {(v,w)};
    if ((v,w) does not form a cycle in T)
        T = T ∪ {(v,w)};
    else
        discard (v,w);
}
if (|T| < n-1)
    printf("no minimum spanning tree\n");
```

In the code, T is the set of chosen edges and E the set of edges not yet tried; $|T|$ denotes the number of chosen edges.

- **Line 1.** At the start no edge has been chosen. Every vertex is its own single-vertex cluster.
- **Line 2.** The loop stops when one of its two conditions fails. If $n - 1$ edges have been chosen, the tree is complete; there is no need to look at more edges. It also stops if no edge is left to choose.
- **Lines 3–4.** The cheapest of the remaining edges is taken and removed from the set. An edge is tried at most once.
- **Lines 5–8.** If the edge does not form a cycle with the edges chosen so far, it is added to the tree; if it does, it is discarded. A discarded edge is never considered again. An edge that forms a cycle with the chosen ones will form a cycle again whatever is chosen afterward.
- **Lines 10–11.** If fewer than $n - 1$ edges have been chosen when the loop ends, the edges have run out but the tree could not be completed. This shows that the graph is not connected, and there is no minimum spanning tree.

The intermediate states of Kruskal's algorithm are a forest, not a tree. The chosen edges can form several disjoint **clusters** (connected components). Clusters grow by merging, and in the end a single cluster remains. That an edge forms a cycle means that its two endpoints are already in the same cluster.

17.3.2 Kruskal step by step

When the nine edges of the seven-village graph are sorted by cost, this sequence results: (0,5) 10, (2,3) 12, (1,6) 14, (1,2) 16, (3,6) 18, (3,4) 22, (4,6) 24, (4,5) 25, (0,1) 28. Table 17.2 shows the decision for each edge, and Figure 17.7 shows four selected moments.

Step	Edge	w	Clusters of the endpoints	Decision	Clusters	Total
1	(0,5)	10	{0} and {5}	add to tree	{0,5}	10
2	(2,3)	12	{2} and {3}	add to tree	{0,5} {2,3}	22
3	(1,6)	14	{1} and {6}	add to tree	{0,5} {2,3} {1,6}	36
4	(1,2)	16	{1,6} and {2,3}	add to tree	{0,5} {1,2,3,6}	52
5	(3,6)	18	both in {1,2,3,6}	cycle, discard	{0,5} {1,2,3,6}	52
6	(3,4)	22	{1,2,3,6} and {4}	add to tree	{0,5} {1,2,3,4,6}	74
7	(4,6)	24	both in {1,2,3,4,6}	cycle, discard	{0,5} {1,2,3,4,6}	74
8	(4,5)	25	{1,2,3,4,6} and {0,5}	add to tree	{0,1,2,3,4,5,6}	99
—	(0,1)	28	never examined: 6 edges have been chosen			

Table 17.2: Kruskal on the seven-village graph. The “Clusters” column is the state after the decision; singleton clusters are not written.

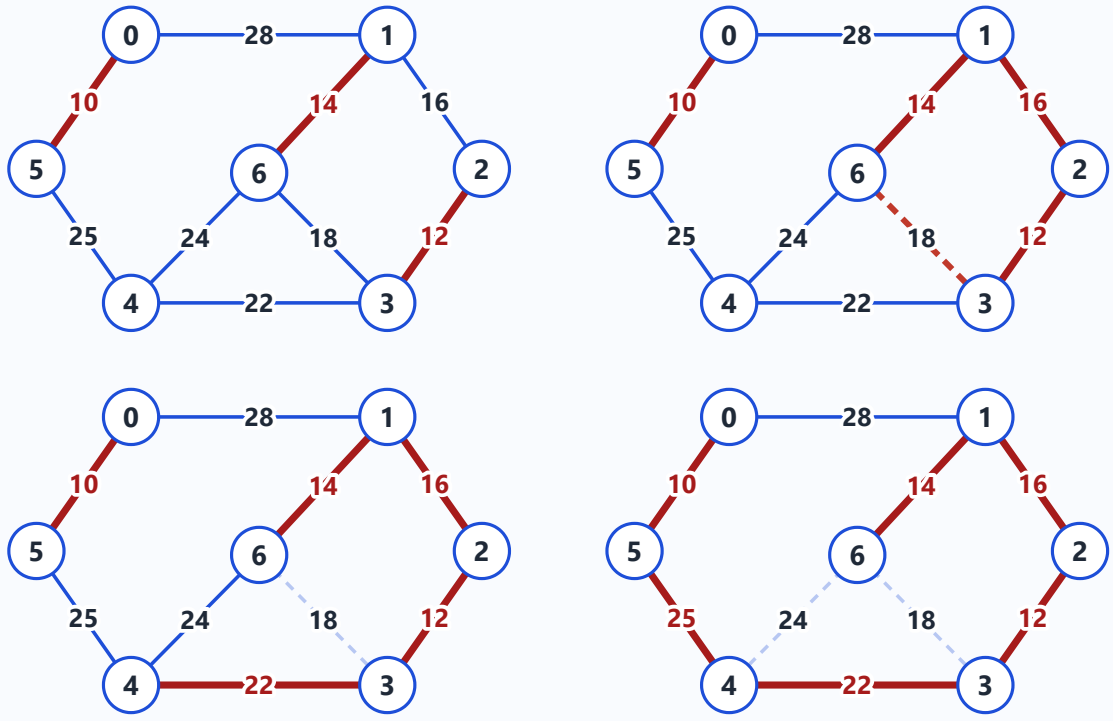


Figure 17.7: Four moments of Kruskal's algorithm. Top left: after the third step there are three separate clusters: {0,5}, {2,3}, {1,6}. Top right: (3,6) is drawn as a red dashed line because it forms a cycle, and it is discarded. Bottom left: (3,4) has been added and the total has risen to 74. Bottom right: the tree is complete: six edges, total 99.

Four rows of the table need attention. In the fourth step (1,2) merges two separate clusters, {1,6} and {2,3}. Before this step the number of clusters (singletons included) was 4; afterward it drops to 3. In the fifth step both endpoints of (3,6) are in the same cluster. The path 3-2-1-6 from 3 to 6 already exists in the tree; (3,6) would close this path and form a cycle. In the seventh step the same holds for (4,6): the path 4-3-2-1-6 is ready. In the eighth step the sixth edge is chosen and the algorithm stops. The most expensive edge, (0,1), is never looked at.

The tree found is {(0,5), (2,3), (1,6), (1,2), (3,4), (4,5)} and its cost is *10 + 12 + 14 + 16 + 22 + 25 = 99*. This is the cheapest of the 46 spanning trees.

17.3.3 How not to code Kruskal

Two lines of the pseudocode remain vague: “pick the cheapest edge” and “if it does not form a cycle”. How these two jobs are done determines the speed of the code. In class, the instructor first deliberately goes down a wrong path and then shows the right one. Seeing the wrong path makes it easier to understand the right one.

Someone looking at the cluster counts in the table makes an observation: the clusters first rose to two, then to three, then fell back to two. “I've got it!”, they say, “I'll keep the clusters of the chosen edges in lists.” The unchosen edges are in an array as well. Searching the array for the cheapest edge every time, deleting it and shifting the rest is a bad idea. It is better to sort the edges once by cost and keep them as triples (end1, end2, cost).

With this plan the trace goes as follows. (0,5) arrives: it is searched for in the lists; neither 0 nor 5 is there, and a new list is opened. New lists are opened for (2,3) and (1,6) as well. When (1,2) arrives, 1 is found in one list and 2 in another. Since they are in different lists, there is no cycle; the edge is taken and the two lists are merged. When (3,6) arrives, both are in the same list: a cycle!

This code works correctly. But for each edge both endpoints are searched for in all the lists, and this search takes *O(n)*. The total cost is *O(e · n)*. The instructor is frank at this point: the cycle was caught and the result is correct, but this is not how it is coded, and code written this way earns no points on the exam. The algorithm itself is not wrong; the data structure behind it is. The instructor also says Kruskal's coding could be given as homework, and that they wonder whether students can discover the necessary data structures on their own.

> **Common mistake**
>
> Checking for a cycle with the question “have both endpoints been chosen before?” In the fourth step of the seven-village example both endpoints of (1,2) are on previously chosen edges: 1 on (1,6) and 2 on (2,3). By this rule the edge is rejected. Yet the endpoints are in different clusters and (1,2) is an edge of the MST. The right question is “are the two endpoints *in the same cluster*?”

17.3.4 Two questions, two data structures

At each step of Kruskal's algorithm two questions are asked, and each has a data structure that answers it.

1) Which edge has the least cost? This question points to a structure that asks again and again for the smallest item: a **priority queue** implemented as a **min-heap**. In a min-heap every node is less than or equal to its children; the smallest item is at the root. Taking the root (deleteMin) takes *O(log e)*. A heap can be built in two ways. If the edges are inserted one by one, *O(e log e)* is needed. If the edge array is ready, **bottom-up heap construction** needs only *O(e)*. In this method, every node from the last internal node up to the root is sifted down and the array turns into a heap. The third way is to sort the edges once; in C the standard library function `qsort` is used for this.

2) Does this edge form a cycle? If the two endpoints of the edge are in different clusters, it does not form a cycle. The clusters are **disjoint sets**: every vertex is in exactly one set. Two operations are needed on disjoint sets. **Find** tells which set a vertex is in by returning the root of the set. **Union** gathers two sets into one. This pair is called **union-find** for short. The cycle question thus comes down to a comparison: do `find(3)` and `find(2)` give the same root?

The instructor shares the surprise at this point: heaps and sets have long since been taught in the background, but the student, without noticing that they would be useful in Kruskal's algorithm, goes and tries to keep lists. The sets taught in primary school speed up a graph algorithm here.

The slides give the cost with a detailed formula. If the heap is built by inserting the edges one by one, about $|E| \cdot \lg |E|$ work is needed. Then the smallest edge is taken $|V|$ times; each removal takes $\lg |E|$ and each set check almost constant time. The total is $|E| \cdot \lg |E| + |V| \cdot (\lg |E| + \Theta(1))$. If the heap is built bottom-up, the first term drops to $2|E|$. The “almost constant” time here is the cost of union-find, bounded by the inverse Ackermann function α. The formula is optimistic at one point: edges that form cycles are also taken from the heap. In the worst case the smallest can be taken e times, not $n - 1$. The safe upper bound is *O(e log e)*.

17.3.5 Union-find and the comparison function

The C implementation of Kruskal rests on three small parts: an edge structure, the union-find operations and a comparison function for `qsort`.

```c
typedef struct { int u, v, w; } edge;

int find(int parent[], int i)   /* root: parent < 0 */
{
    while (parent[i] >= 0) i = parent[i];
    return i;
}
void unite(int parent[], int a, int b)   /* a, b are roots */
{
    if (parent[a] <= parent[b]) {   /* a is the bigger set */
        parent[a] += parent[b];  parent[b] = a;
    } else {
        parent[b] += parent[a];  parent[a] = b;
    }
}
int cmp(const void *x, const void *y)
{
    return ((const edge *)x)->w - ((const edge *)y)->w;
}
```

- **Line 1.** Each edge is three integers: the endpoints `u`, `v` and the weight `w`. Thanks to `typedef`, the type name can be used as `edge` without writing `struct`. On x86-64 `sizeof(edge)` is 12 bytes; there is no padding, since all three fields are 4-byte `int`s.
- **Lines 3–7.** `find` finds the root of the set of a vertex. The sets are kept as trees in an array: `parent[i]` is the parent of `i` in the tree. The root has no parent; at a root, `parent` is a negative number. The loop climbs through parents until it reaches a negative value. The returned root is the name of the set.
- **Lines 8–15.** `unite` takes two roots and merges their sets. The parameters must be roots; if it is called with a non-root vertex, the tree is corrupted. The absolute value of the negative number stored at the root is the set size: `parent[a] = -4` says that the set of `a` has 4 elements. The condition `parent[a] <= parent[b]` means that the set of `a` is larger than or equal to the other ($-4 < -2$). The larger set stays as the root: the sizes are added (`parent[a] += parent[b]`) and the root of the smaller set is attached under the larger one (`parent[b] = a`). The `else` branch does the same job in the opposite direction.
- **Lines 16–19.** `cmp` has the signature that `qsort` wants: it takes two `const void *`, and returns a negative number if the first is smaller, a positive one if larger, and zero if equal. The `void *` pointers are converted to `const edge *` and the weights are subtracted.

The in-memory view of the sets is a single integer array. At the start each vertex is the root of its own set and has size 1; the array is `{-1, -1, -1, -1, -1, -1, -1}`. In the seven-village example, after the first four edges are added, the array becomes `{-2, -4, 1, 2, -1, 0, 1}`. This array defines three trees. 0 is a root (size 2) and is the parent of 5. 1 is a root (size 4); 2 and 6 are attached directly to 1, and 3 to 2. 4 is alone. The call `find(parent, 3)` follows the path 3 → 2 → 1 and returns 1.

Theorem 17.4 (Weighting rule)

If in a union the smaller set is always attached under the larger (the **weighting rule**), the height of the tree of a set with m elements is at most $\lfloor \lg m \rfloor + 1$. So `find` takes $O(\log n)$.

Proof. The depth of a vertex grows by one only when its own set is attached under a set that is larger or equal. At that moment the size of the set containing the vertex at least doubles. The size starts at 1 and cannot exceed m; doubling can happen at most $\lfloor \lg m \rfloor$ times. If the depth of the root is counted as 1, the height is at most $\lfloor \lg m \rfloor + 1$. ■

Without the rule, in the worst case the trees would turn into chains. If, in the form `unite(0, 1)`, `unite(0, 2)`, …, the large set were attached under the small one each time, `find` would take $O(n)$ steps.

One more improvement is possible: **path compression**. In the book of Horowitz, Sahni and Anderson-Freed the same idea goes by the name collapsing rule. After finding the root, `find` attaches every vertex on the path it climbed directly to the root. Later searches pass the same path in one step:

```
int find(int parent[], int i)       /* with path compression */
{
    int root = i, next;
    while (parent[root] >= 0) root = parent[root];     /* pass 1: find the root */
    while (i != root) {                          /* pass 2: hook the path vertices to the
root */
        next = parent[i];
        parent[i] = root;
        i = next;
    }
    return root;
}
```

The first loop finds the root and the second loop walks the path once more. The old parent of each vertex is saved in `next`, and then the vertex is attached to the root. Without the saving, the rest of the path would be lost. When the weighting rule and path compression are used together, the amortized cost per operation over a sequence of operations is $O(\alpha(n))$. α is the inverse Ackermann function and does not exceed 4 even for n as large as the number of atoms in the universe; in practice it is constant. On a random graph with 2000 vertices (8000 edges), the number of links climbed by Kruskal's `find` calls was measured as 21,464 without path compression and 12,348 with it.

Common mistake

Writing `unite(parent, k[i].u, k[i].v)`, that is, uniting the endpoints of the edge directly. `unite` expects two roots. If an endpoint is not a root, its `parent` value points to a parent; the function takes this value for a set size and the array is corrupted. The roots are first found with `find`, and the union is done between the roots.

The subtraction in the `cmp` function also hides a trap. The difference of two large weights can exceed the `int` limit; for example *2,000,000,000 − (−2,000,000,000)* overflows and the sign comes out wrong. Here the weights are small nonnegative integers and no overflow occurs. In a general-purpose comparison the form `(a > b) - (a < b)` is safe.

17.3.6 The main loop of `kruskal()` and its trace

With the parts ready, the main loop fits in eleven lines. In the code, `k` is the edge array of `e` items; `T` is the array of *n − 1* items in which the chosen edges will be written; `parent` is the set array of `n` items. `i` and `t` are integers.

```
for (i = 0; i < n; i++) parent[i] = -1;
qsort(k, e, sizeof(edge), cmp);
for (i = 0, t = 0; i < e && t < n - 1; i++) {
    int a = find(parent, k[i].u);
    int b = find(parent, k[i].v);
    if (a != b) {                 /* no cycle */
        T[t++] = k[i];
        unite(parent, a, b);
    }                             /* otherwise discard */
}
if (t < n - 1) printf("no tree\n");
```

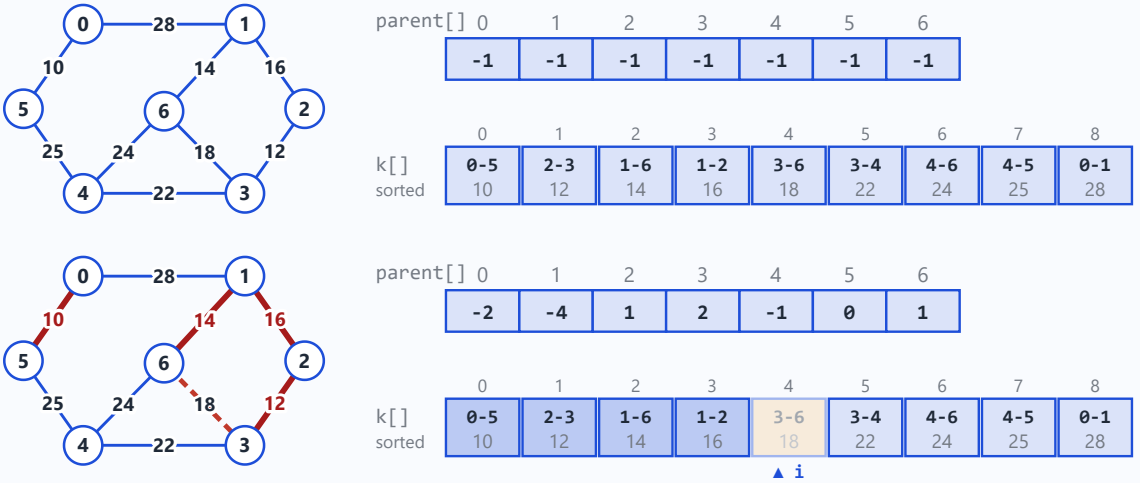
- **Line 1.** Every vertex is made the root of its own set; since its size is 1, its `parent` value is −1. The cost is *O(n)*.
- **Line 2.** The edges are arranged in increasing order of cost. `qsort` is given the start of the array, the number of items, the size in bytes of one item and the comparison function. `qsort` does not know the type of the items; with `sizeof(edge)` it learns how many bytes each item has and moves items byte by byte. The cost is *O(e log e)*.
- **Line 3.** Starting from the beginning of the array, edges are tried. Two counters are initialized together: `i` is the index of the edge being tried and `t` is the number of chosen edges. The loop stops when the edges run out or when *n − 1* edges have been chosen. If the second condition is forgotten, the algorithm still gives the right result, but all the remaining edges are tried in vain after the tree is complete.

- **Lines 4–5.** The roots of the two endpoints of the edge are found. `a` and `b` are declared in the loop body; since C99 it is allowed to declare variables inside a block.
- **Lines 6–9.** If the roots differ, the endpoints are in separate sets and the edge does not form a cycle. The edge is copied into `T` (structure assignment copies all three fields at once), `t` grows by one and the two sets are united. If the roots are the same, nothing is done; the edge is discarded.
- **Line 11.** If the loop ended while $t < n - 1$, the edges have run out; the graph is not connected.

The code was compiled in the form given in the slides and run with a test driver. Table 17.3 is the program's own output: the roots found for each edge and the `parent` array after the union.

`i`	`k[i]`	`a`	`b`	**Decision**	`t`	`parent[]`
—	—	—	—	start	0	`-1 -1 -1 -1 -1 -1 -1`
0	(0,5,10)	0	5	add	1	`-2 -1 -1 -1 -1 0 -1`
1	(2,3,12)	2	3	add	2	`-2 -1 -2 2 -1 0 -1`
2	(1,6,14)	1	6	add	3	`-2 -2 -2 2 -1 0 1`
3	(1,2,16)	1	2	add	4	`-2 -4 1 2 -1 0 1`
4	(3,6,18)	1	1	discard	4	unchanged
5	(3,4,22)	1	4	add	5	`-2 -5 1 2 1 0 1`
6	(4,6,24)	1	1	discard	5	unchanged
7	(4,5,25)	1	0	add	6	`1 -7 1 2 1 0 1`
8	—	condition `t < n - 1` false: the loop ends; cost 99				

Table 17.3: Trace of the main loop of `kruskal()` on the seven-village graph. `a`, `b`: the roots found in lines 4–5. The last column is `parent[0..6]` after the decision.



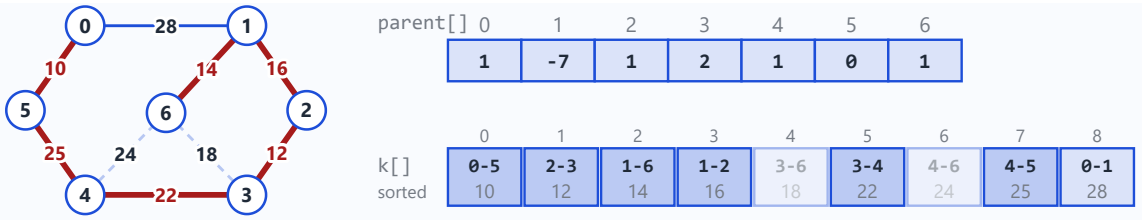


Figure 17.8: Three frames from the trace of `kruskal()`. Top: the start, every `parent` is −1 and the edges are sorted. Middle: `i` = 4: both endpoints of (3,6) have root 1, the edge forms a cycle and is discarded. Bottom: the final state: the root 1 holds the single set of seven vertices (`parent[1]` = −7); the faint boxes are the discarded edges.

A few details of the trace show how the weighting rule works. At `i` = 0 both sets have 1 element; `parent[0] <= parent[5]` is true by equality, so 0 stays the root. At `i` = 3, {1,6} and {2,3} are again of equal size; 1 stays the root, 2 goes under it, and the size becomes −4. From this point on, finding the root of 3 takes two steps: 3 → 2 → 1. At `i` = 7, the set of 1 has five elements and that of 0 has two; the small set (0) is attached under the large one (1) and `parent[0]` becomes 1. In the final array the deepest vertices are 3 and 5; both are two links from the root. For a set of seven elements, the bound given by Theorem 17.4 is $\lfloor \lg 7 \rfloor + 1 = 3$ levels; the tree is within this bound.

The same driver tried two more cases. On an exam graph with distinct weights the cost came out as 16. On a disconnected graph (two components, five vertices) only three edges were chosen and the program printed “no tree”. In all the experiments a wrapper that counts `malloc` and `free` calls counted 12 allocations and 12 frees; there is no leak.

17.3.7 The complexity of Kruskal's algorithm

The cost of Kruskal's algorithm is obtained by adding up how many times each line of the code runs and how long each run takes. Table 17.4 gives this sum by groups of lines.

Step	Cost	Explanation
Sorting or building the heap	$O(e \log e)$ or $O(e)$	`qsort` or bottom-up heap construction
Taking the smallest edge	$O(\log e)$, at most e times	deleting the root in a min-heap; $O(1)$ in a sorted array
`find` (twice per edge)	$O(\log n)$ with the weighting rule; $\approx O(\alpha(n))$ with path compression	at most $2e$ calls
`unite`	$O(1)$	exactly $n - 1$ times
Total	$O(e \log e) = O(e \log n)$	since $e \le n^2$, $\log e \le 2 \log n$

Table 17.4: The cost of Kruskal's algorithm with union-find.

In the total, the dominant term is the sorting. The `find` calls cost at most $2e \cdot O(\log n)$, which is also $O(e \log n)$. The equality in the last row rests on a simple observation. In a simple graph $e \le n(n-1)/2 < n^2$. Taking logarithms gives $\log e < 2 \log n$; the constant factor drops out in O notation and $O(e \log e)$

$= O(e \log n)$. In a connected graph $e \geq n - 1$, so in the other direction $\log n = O(\log e)$ also holds. The memory cost is $O(n + e)$ for the edge array, `T` and `parent`.

In the best case none of the first $n - 1$ edges forms a cycle and the loop ends after $n - 1$ rounds. But the sorting still processes all the edges, and a comparison-based sort in general takes $O(e \log e)$; the early completion of the tree does not reduce this cost. If a heap is used, the table changes. The heap is built in $O(e)$, and then only as many smallest edges as needed are taken. If the tree is completed early, the total is $O(e + k \log e)$, where k is the number of edges tried. This is why a heap is asked for in the homework. On a dense graph with random weights (2000 vertices, about two million edges) the tree was completed after only 9685 edges had been tried; `qsort`, however, sorted all two million edges.

The comparison with the "bad" coding that keeps lists sums up the main idea of the chapter. In the bad coding, for every edge the endpoints are searched for in all the lists: $O(n)$ per edge, $O(e \cdot n)$ in total. For seven villages the difference is negligible. On a graph with 5000 vertices and six million edges, however, $e \cdot n$ exceeds thirty billion while $e \log e$ is about a hundred and forty million. The algorithm is the same; only the data structure behind it changes.

17.4 Prim's algorithm

17.4.1 The algorithm

The history of Prim's algorithm is interesting. The Czech mathematician Vojtěch Jarník published the algorithm in 1930. Robert C. Prim in 1957 and Edsger W. Dijkstra in 1959 rediscovered the same algorithm independently of one another. That is why it also appears in sources as the DJP algorithm or the Prim–Jarník algorithm.

In Kruskal's algorithm the chosen edges could form disjoint clusters. Prim's algorithm, on the other hand, grows a single tree. It starts with a start vertex and at each step adds the cheapest edge that connects a vertex in the tree to a vertex outside the tree. The pseudocode is as follows:

```
T  = {};
TV = {0};           /* tree vertices: start from 0 */
while (|T| < n-1) {
    pick the cheapest edge (u,v) with u ∈ TV and v ∉ TV;
    if (no such edge) break;
    TV = TV ∪ {v};
    T  = T  ∪ {(u,v)};
}
if (|T| < n-1)
    printf("no minimum spanning tree\n");
```

- **Lines 1–2.** T is the set of chosen edges and TV the set of vertices that have joined the tree. The start vertex is 0; any vertex can be chosen.

- **Lines 3–4.** In each round the cheapest edge with one endpoint in *TV* and the other outside *TV* is sought. Only these are the edges that can be chosen.
- **Line 5.** If there is no such edge, the tree cannot reach the remaining vertices; the graph is not connected.
- **Lines 6–7.** The outer endpoint of the edge is added to *TV* and the edge itself to *T*.
- **Lines 9–10.** If the loop ended early, there is no minimum spanning tree.

No cycle check is needed in Prim's algorithm. One endpoint of the chosen edge is in the tree and the other is outside; the condition $v \notin TV$ already guarantees that the edge goes to a new vertex. An edge that goes to a new vertex cannot form a cycle. The chosen edge is the cheapest edge crossing the cut $S = TV$; the cut property says that every choice is correct.

17.4.2 Prim step by step

Let Prim be started from 0 on the seven-village graph. Table 17.5 gives *TV*, the edges that can be chosen and the choice in each round.

Round	TV (at start of round)	Candidate edges	Chosen	Total
1	{0}	(0,1) 28, (0,5) 10	(0,5)	10
2	{0, 5}	(0,1) 28, (5,4) 25	(5,4)	35
3	{0, 5, 4}	(0,1) 28, (4,3) 22, (4,6) 24	(4,3)	57
4	{0, 5, 4, 3}	(0,1) 28, (3,2) 12, (3,6) 18, (4,6) 24	(3,2)	69
5	{0, 5, 4, 3, 2}	(0,1) 28, (2,1) 16, (3,6) 18, (4,6) 24	(2,1)	85
6	{0, 5, 4, 3, 2, 1}	(1,6) 14, (3,6) 18, (4,6) 24	(1,6)	99

Table 17.5: Prim from 0 on the seven-village graph. The “Candidates” column holds the edges with one endpoint in TV and the other outside.

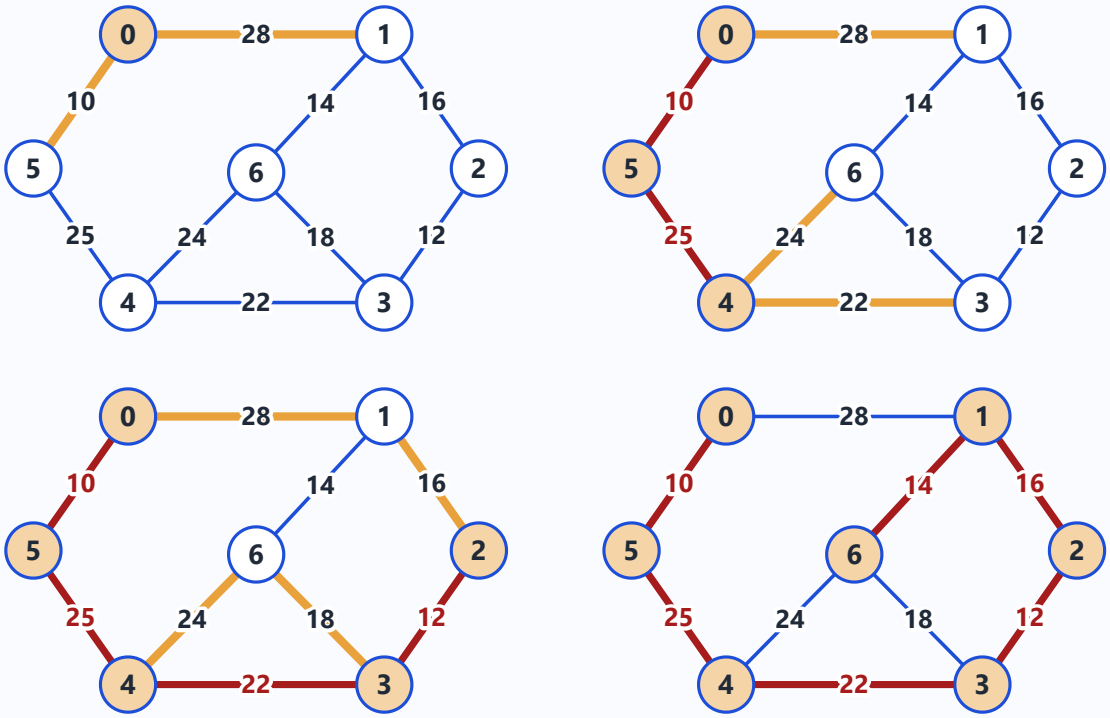


Figure 17.9: Four moments of Prim's algorithm. Filled vertices are in TV, thick yellow edges are the candidate edges of that round, and thick red edges are in the tree. Top left: round 1. Top right: round 3. Bottom left: round 5. Bottom right: the tree is complete, total 99.

The “candidates” column of the table changes in every round. When a vertex enters the tree, the edges touching it join the candidates. When both endpoints of an edge are in the tree, that edge drops out of the candidates. For example, (0,1) is a candidate for five rounds but a cheaper candidate is always found. At the start of the sixth round 1 is also in the tree; both endpoints of (0,1) are in *TV* and it is no longer a candidate. The slides draw attention to this point separately: (0,1) still looks “choosable” in the drawing, but if it were chosen it would form a cycle.

The tree Prim found is {(0,5), (5,4), (4,3), (3,2), (2,1), (1,6)} and is the same as Kruskal's; only the order in which the edges were chosen differs. This is not a coincidence. In the seven-village graph all weights are distinct and the minimum spanning tree is unique. Changing the start vertex does not change the result: Prim started from 3 chooses the edges in the order (3,2), (2,1), (1,6), (3,4), (4,5), (5,0) and again arrives at 99.

17.4.3 The real difficulty of coding Prim

The fourth line of the pseudocode is a single sentence but is not easy to code: “pick the cheapest edge with one endpoint in the tree and the other outside”. This line runs in every round and carries the following requirements:

- The candidate edge with the least cost must be obtainable in one move.

- When an edge is chosen, new edges are added to the candidates and edges with both endpoints in the tree are removed. The candidate set changes in every round.
- When 0 is chosen, a structure is needed that sees only (0,1) and (0,5); it should not even look at the rest of the edges of the graph.

A naive coding scans all the edges in every round and looks for the cheapest among those with an endpoint in the tree. This costs *O(e)* per round, *O(n · e)* in total. It is the Prim counterpart of the list mistake in Kruskal's algorithm.

The instructor stresses the importance of the topic with two observations. First: a student who can find an efficient solution for this line has reached a certain level in data structures. Someone who says "Prim has already written it, I'll just code it" and translates the pseudocode directly into code sees the complexity suddenly grow. Second: if this problem were given to a graph theorist who had never heard the word "algorithm", they too would do what Prim did: choose the cheapest edge. What sets the computer engineer apart is discovering which data structure must stand behind this choice.

The solution rests on this observation: it is not necessary to keep all the candidate edges. For each vertex *v* outside the tree a single number is enough: the cost of the cheapest edge connecting *v* to the tree. This number is kept in the array `lowcost[v]`, and the tree end of that edge in the array `closest[v]`. The cheapest candidate edge is the edge of the vertex outside the tree with the smallest `lowcost` value. When a vertex *u* enters the tree, only the `lowcost` values of the neighbors of *u* can change. This idea gives two implementations: $O(n^2)$ with arrays and *O(e log n)* with a min-heap.

17.4.4 The $O(n^2)$ implementation with arrays

The graph is kept as a cost matrix: `w[u][v]` is the weight of the edge *(u, v)*. If there is no edge, `w[u][v]` is a large constant `INF`; the diagonal is 0. `in_tree[v]` says whether *v* is in the tree ($v \in TV$). The C code of Prim in the slides is as follows:

```
in_tree[0] = 1;                    /* TV = {0} */
for (v = 0; v < n; v++)
    lowcost[v] = w[0][v], closest[v] = 0;
for (k = 1; k < n; k++) {
    u = cheapest(lowcost, in_tree, n);  /* O(n) */
    if (u < 0) break;                  /* no tree */
    in_tree[u] = 1;  total += lowcost[u];
    for (v = 0; v < n; v++)
        if (!in_tree[v] && w[u][v] < lowcost[v])
            lowcost[v] = w[u][v], closest[v] = u;
}
```

- **Line 1.** 0 is taken into the tree. The other `in_tree` items and `total` must have been set to 0 beforehand; the code fragment assumes this.

- **Lines 2–3.** At the start only 0 is in the tree. So the cheapest connection of each *v* to the tree is directly the edge *(0, v)*: `lowcost[v] = w[0][v]`, `closest[v] = 0`. For vertices with no edge, `lowcost` stays `INF`. The comma operator performs two assignments in a single statement.
- **Line 4.** *n − 1* more vertices will be added to the tree; the loop runs *n − 1* rounds.
- **Lines 5–6.** `cheapest` returns, among the vertices outside the tree, the one with the smallest `lowcost` value. If all `lowcost` values are `INF`, no vertex is left that can be connected to the tree; the function returns −1 and the loop is broken.
- **Line 7.** `u` is taken into the tree; its edge is *(closest[u], u)* and its cost is added to the total.
- **Lines 8–10.** The vertex `u` that has just entered the tree may offer a cheaper connection to the vertices outside. For each *v* outside the tree, if `w[u][v]` is smaller than the recorded `lowcost[v]`, the record is updated: now the cheapest connection of *v* is through `u`.

The function `cheapest` is not given in the slides. Its implementation is a simple linear search:

```
int cheapest(const int lowcost[], const int in_tree[], int n)
{
    int v, u = -1;
    for (v = 0; v < n; v++)
        if (!in_tree[v] && lowcost[v] < INF &&
            (u < 0 || lowcost[v] < lowcost[u]))
            u = v;
    return u;            /* -1: no vertex left connected to TV */
}
```

The search looks only at vertices outside the tree that can be connected to it (whose `lowcost` value is not `INF`). The condition `u < 0` makes the first suitable vertex the candidate; every later vertex with a smaller `lowcost` value takes the candidate's place.

k	u	Edge entering the tree	total	Updated (`lowcost` , `closest`)
—	—	start	0	`lowcost` = 0 28 ∞ ∞ ∞ 10 ∞; `closest` all 0
1	5	(0,5) 10	10	lowcost[4] = 25 (5)
2	4	(5,4) 25	35	lowcost[3] = 22 (4), lowcost[6] = 24 (4)
3	3	(4,3) 22	57	lowcost[2] = 12 (3), lowcost[6] = 18 (3)
4	2	(3,2) 12	69	lowcost[1] = 16 (2)
5	1	(2,1) 16	85	lowcost[6] = 14 (1)
6	6	(1,6) 14	99	none
7	—	loop ends; `lowcost` = 0 16 12 22 25 10 14, `closest` = 0 2 3 4 5 0 1		

Table 17.6: Trace of `prim()` (seven villages, from 0). The “Updated” column holds the records changed by lines 8–10. The arrays in the last row are the state when the loop ends.

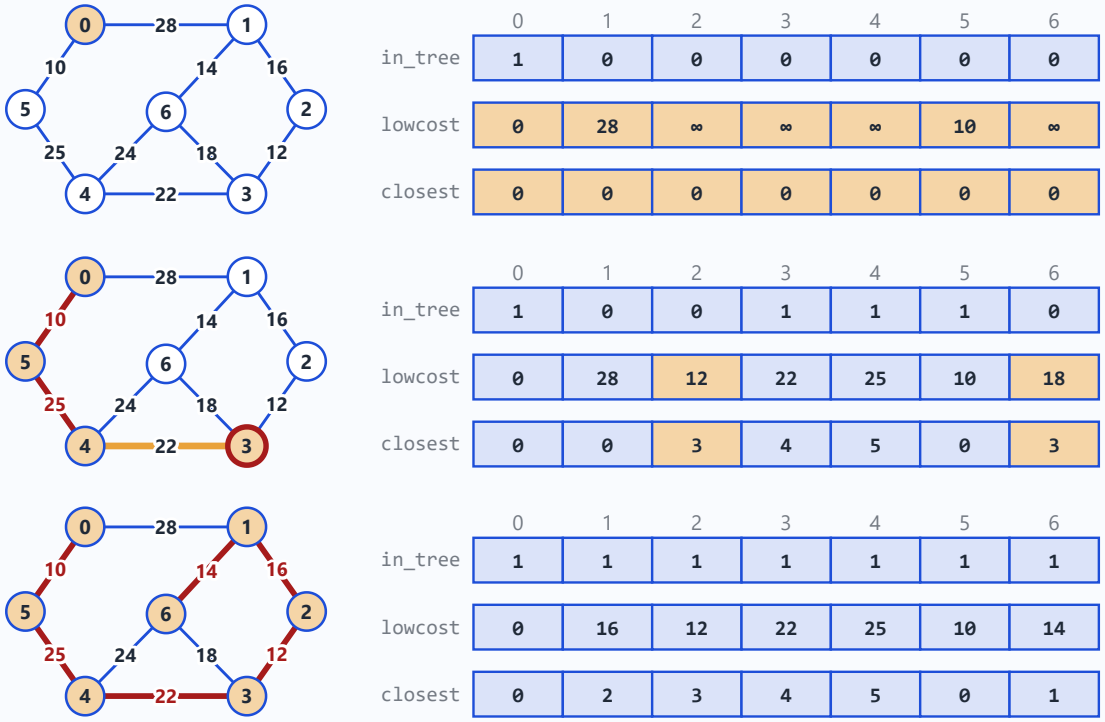


Figure 17.10: Three frames from the trace of `prim()`. Top: the start: only 1 (28) and 5 (10) are connected to 0. Middle: after 3 has entered the tree: lowcost[2] = 12 and lowcost[6] = 18 have been updated. Bottom: the final state: the `closest` array holds the edges of the tree and the `lowcost` array their costs.

The two arrays in the final state are the tree itself. For every $v \neq 0$, *(closest[v], v)* is a tree edge and `lowcost[v]` is its cost: (2,1) 16, (3,2) 12, (4,3) 22, (5,4) 25, (0,5) 10, (1,6) 14. The sum of `lowcost` except for its first item is 99. The record of 6 changed three times: first 24 (through 4), then 18 (through 3), and last 14 (through 1). Each update is a vertex that has just entered the tree opening a cheaper route to 6.

Cost. The outer loop runs $n - 1$ rounds. In each round `cheapest` looks at n vertices and the update loop scans n columns. That is $O(n)$ per round, $O(n^2)$ in total. This time is independent of the number of edges; the best and worst cases are the same: $\Theta(n^2)$. Memory is $O(n^2)$ for the matrix and $O(n)$ for the arrays. In a dense graph, that is when $e \approx n^2$, looking at all the edges at least once already requires $\Omega(n^2)$. So this version is ideal for dense graphs.

The code was compiled in the form given in the slides and run; the values in Table 17.6 are the program's output. On a disconnected graph `cheapest` returned −1 in the third round, the loop was broken and only two edges were chosen.

Common mistake

Using `INT_MAX` for infinity and getting overflow in sums. The comparison `w[u][v] < lowcost[v]` is safe with `INT_MAX`. But if a vertex that cannot be connected to the tree is chosen by mistake, `total += lowcost[u]` overflows. If the same code is adapted to problems

such as shortest paths and `d[u] + w[u][v]` is computed, the sum overflows directly; signed integer overflow is undefined behavior in C. The safe way is to choose a constant that is larger than all possible sums but whose double still fits in an `int`; here `INF` = 1,000,000 was used. In addition, the condition `lowcost[v] < INF` in `cheapest` prevents a vertex that cannot be connected from being chosen.

Common mistake

Forgetting the condition `!in_tree[v]` in the update loop. The `lowcost` and `closest` values of a vertex in the tree record the edge by which it entered and must never change again. Without the condition, the first corruption comes immediately. In the round when `u` enters the tree, for `v = u` the value `w[u][u] = 0` is smaller than the recorded value. So `lowcost[u]` becomes 0 and `closest[u]` becomes `u`. This version was tried on the seven-village graph. `total` still came out as 99 because it was added before the record was corrupted; but when the loop ended, the `closest` array was 0 1 2 3 4 5 6. Every vertex appeared linked to itself and none of the tree's edges was left. If the diagonal were `INF` instead of 0, this time the neighbors in the tree would corrupt each other's records. The general name of this mistake is not removing from the candidate set the edges that have both endpoints in the tree.

17.4.5 Speeding up with a heap

The array version scans n vertices in every round. In a sparse graph, for example when $e \approx 4n$, most of these scans are wasted: the `lowcost` value of most vertices outside the tree is still infinite. Two improvements are made together. The graph is kept with adjacency lists; that way, when a vertex enters the tree, only its real neighbors are looked at. The smallest `lowcost` value is also taken not by a linear search but from a min-heap. The code in the slides is as follows:

```
/* adjacency list + min-heap (key = lowcost) */
heap_push(H, 0, 0);
while (!heap_empty(H)) {
    u = heap_pop_min(H);          /* O(log n) */
    if (in_tree[u]) continue;     /* stale entry */
    in_tree[u] = 1;
    for (p = graph[u]; p; p = p->link)
        if (!in_tree[p->vertex] &&
            p->weight < lowcost[p->vertex]) {
            lowcost[p->vertex] = p->weight;
            closest[p->vertex] = u;
            heap_push(H, p->vertex, p->weight);
        }
}
```

In the adjacency list, each vertex has a list; `graph[u]` points to the first item of the list of `u`. Each item has three fields: the neighbor's number `vertex`, the weight of the edge `weight` and `link`, which points to the next item in the list. Each record of the heap is a (vertex, key) pair, and the key is the `lowcost` value at the time the record was added. Before the code runs, all `lowcost` values must be set to `INF`, `lowcost[0]` to 0 and all `in_tree` values to 0.

- **Line 2.** The start vertex 0 is put into the heap with key 0.
- **Lines 3–4.** While the heap is not empty, the record with the smallest key is taken. `heap_pop_min` returns the root, moves the last record to the root and sifts it down: *O(log m)*, where *m* is the number of records in the heap.
- **Line 5.** If the vertex taken is already in the tree, the record is **stale** and is skipped. A stale record arises like this: when the `lowcost` value of a vertex decreases, the old record is not deleted from the heap; a second record with the new key is added. The record with the smaller key comes out first and the vertex enters the tree. When the old record comes out later, the vertex is already marked.
- **Line 6.** The vertex is taken into the tree. The edge by which it entered is *(closest[u], u)* and its cost is `lowcost[u]`.
- **Lines 7–13.** For each neighbor in the list of `u`, the update of the array version is performed. If the neighbor is outside the tree and the edge *(u, neighbor)* is cheaper than the recorded connection, `lowcost` and `closest` are updated and a heap record with the new key is added. The list walk is the familiar pattern for linked lists, `for (p = graph[u]; p; p = p->link)`; the loop ends when `p` becomes `NULL`.

The slides do not give the inside of the heap operations. With a binary min-heap built on an array, they can be written as follows:

```c
typedef struct { int vertex, key; } entry;
typedef struct {
    entry *a;          /* a[1..size]: root a[1] */
    int size, capacity;
} heap;

int heap_empty(const heap *H) { return H->size == 0; }

void heap_push(heap *H, int vertex, int key)
{
    int i = ++H->size;                    /* capacity >= 2e + 1 */
    while (i > 1 && H->a[i / 2].key > key) {
        H->a[i] = H->a[i / 2];             /* move the parent down */
        i /= 2;
    }
    H->a[i].vertex = vertex;  H->a[i].key = key;
}

int heap_pop_min(heap *H)
{
    int root = H->a[1].vertex;
    entry last = H->a[H->size--];
    int i = 1, c = 2;
    while (c <= H->size) {
        if (c < H->size && H->a[c + 1].key < H->a[c].key) c++;
        if (last.key <= H->a[c].key) break;
        H->a[i] = H->a[c];                 /* move the smaller child up */
        i = c;  c *= 2;
    }
    H->a[i] = last;
    return root;
}
```

The records are in the range `a[1..size]`; the parent of `a[i]` is `a[i/2]` and its children are `a[2i]` and `a[2i+1]`. `heap_push` puts the new record at the end of the array and moves it up as long as its key is smaller than its parent's; instead of swapping, it moves the parent down and shifts the hole up. `heap_pop_min` saves the root, takes the last record into a temporary variable and sifts it down from the root until it is no larger than the smaller child. The heap array is allocated with `malloc` for *2e + 2* records. Each adjacency item can cause at most one insertion; there are *2e* items in the lists in total, and with the initial record *2e + 1* records must fit. `a[0]` is not used. The array is released with `free` when the job is done; the *2e* items of the adjacency list must also be freed one by one.

Taken	State	Records added	Heap
—	start	0:0	0:0
0:0	enters the tree	5:10, 1:28	5:10, 1:28
5:10	enters the tree	4:25	4:25, 1:28
4:25	enters the tree	3:22, 6:24	3:22, 6:24, 1:28

Taken	State	Records added	Heap
3:22	enters the tree	2:12, 6:18	2:12, 6:18, 6:24, 1:28
2:12	enters the tree	1:16	1:16, 6:18, 6:24, 1:28
1:16	enters the tree	6:14	6:14, 6:18, 6:24, 1:28
6:14	enters the tree	—	6:18, 6:24, 1:28
6:18	stale, skipped	—	6:24, 1:28
6:24	stale, skipped	—	1:28
1:28	stale, skipped	—	empty

Table 17.7: Trace of Prim with a heap on the seven-village graph. The heap column shows the records after lines 7–13 finish, in key order, in the form “vertex:key”.

The order in which vertices enter the tree is the same as in the array version: 0, 5, 4, 3, 2, 1, 6. In total 10 records were added and 10 records were taken; three were stale. Three records were created for 6 (24, 18, 14), because the `lowcost` value of 6 decreased three times. The program gave these numbers and the total of 99 exactly. The counting `malloc` / `free` wrapper counted 19 allocations (18 list items and the heap array) and 19 frees.

Cost. Each adjacency item makes at most one insertion; at most $2e + 1$ records enter the heap and as many come out. Since there are at most $2e + 1$ records in the heap, each operation takes $O(\log e)$, which is $O(\log n)$. Walking the adjacency lists costs $O(n + e)$ in total. The total is $O((n + e) \log n)$, which is $O(e \log n)$ in a connected graph. Memory is $O(n + e)$.

Instead of leaving stale records, the key of the record in the heap can be decreased in place. This operation is called **decrease-key**. For this, the position of each vertex in the heap array must be kept in a separate array. The heap then holds at most n records, but the asymptotic bound does not change. A Fibonacci heap does decrease-key in amortized $O(1)$ and brings the bound down to $O(e + n \log n)$. This bound is important theoretically; in practice the constants of the structure are large and a binary heap is often faster.

Implementation	Time	Where it fits
Prim, arrays (`lowcost` , `closest`)	$O(n^2)$	dense graph ($e \approx n^2$)
Prim, binary heap + adjacency list	$O(e \log n)$	sparse graph
Prim, Fibonacci heap	$O(e + n \log n)$	theoretical
Kruskal	$O(e \log e)$	sparse graph, edge list ready

Table 17.8: Prim implementations and Kruskal.

To see what the bounds in the table mean in practice, the three implementations were run on random weighted graphs. Table 17.9 gives the times. Times vary with the machine, the compiler and the random graph; what matters here are the ratios.

n	*e*	Graph	Kruskal (qsort)	Prim, arrays	Prim, heap
2000	8000	sparse	0,001	0,017	0,002
2000	1.999.000	dense	0,42	0,019	0,040
5000	20.000	sparse	0,004	0,106	0,003
5000	6.248.750	dense	1,47	0,114	0,109

Table 17.9: Times measured on random connected graphs (seconds; single measurement, zig cc -O2, varies by machine). In all four rows the cost found by the three algorithms is the same.

On sparse graphs, Kruskal and Prim with a heap finish in a few milliseconds; Prim with arrays is 8 to 35 times slower than they are. When the size goes from 2000 to 5000, the time of Prim with arrays grew about six times; this is close to the expectation $(5000/2000)^2 \approx 6.3$. On dense graphs the table turns around. Prim with arrays is unaffected by the number of edges and is among the fastest. Almost all of Kruskal's time goes to sorting millions of edges, although the tree is completed within the first ten thousand edges. On a dense graph Prim with a heap only catches up with the array version. A heap operation is far more expensive than looking at a cell in a matrix, and in a dense graph the number of edges is close to the number of cells.

17.5 Other algorithms and comparisons

17.5.1 Sollin (Borůvka) and reverse-delete

The Czech engineer Otakar Borůvka solved the minimum spanning tree problem in 1926 while designing the electric grid of Moravia. This is the first known algorithm for the problem; since Sollin proposed the same method again in the 1960s, it is known by both names. The algorithm works in rounds. At the start every vertex is a separate component. In each round *every component* picks the cheapest edge leaving it; all the chosen edges are added at the same time and the components merge.

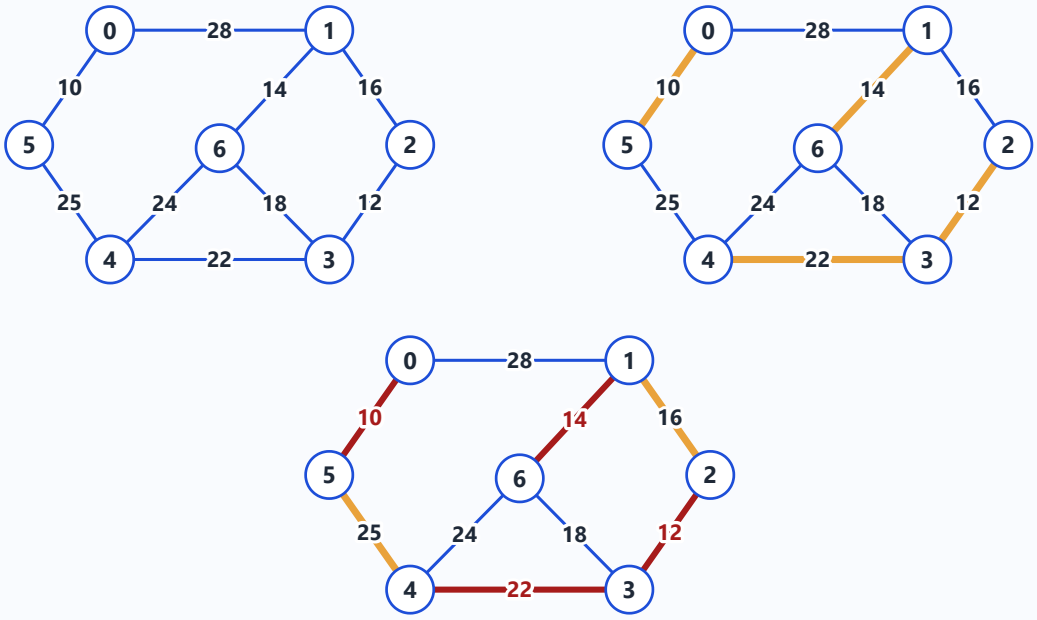


Figure 17.11: Sollin (Borůvka) on the seven-village graph. Left: the start, seven components. Middle: round 1: (0,5), (1,6), (2,3), (3,4) are chosen, three components remain. Right: round 2: (4,5) and (1,2) are chosen, one component remains. Total 99.

In the first round, 0 and 5 share the same cheapest edge: (0,5) = 10. For 1 and 6 it is (1,6) = 14, for 2 and 3 it is (2,3) = 12. The edges of 4 are 22, 24 and 25; the cheapest is (3,4). If two components choose the same edge, the edge is added once. At the end of the round {0,5}, {1,6} and {2,3,4} remain. In the second round the edges leaving {0,5} are (0,1) 28 and (4,5) 25; the cheapest is (4,5). The cheapest for {1,6} is (1,2) 16. The cheapest for {2,3,4} is also (1,2). When the two chosen edges are added, a single component remains.

In every round each component merges with another component; the number of components at least halves. One can get from n components to one in at most $lg\ n$ rounds. Each round looks at all the edges once: $O(e)$. The total is $O(e \log n)$. Since the choices within a round are independent of one another, the work can be divided among many processors; the algorithm is suited to parallelization. If there are ties in the weights, a consistent selection rule among edges of the same weight (for example the endpoint with the smaller number) is needed; otherwise a cycle can form in a round.

Reverse-delete is the reverse of Kruskal's. It starts with all the edges and tries edges in decreasing order of cost. If deleting an edge does not disconnect the graph, the edge is deleted; if it does, the edge is kept. In the seven-village graph (0,1) 28 is deleted; (4,5) 25 is kept, because if it were deleted {0,5} would be cut off. (4,6) 24 and (3,6) 18 are deleted; each of the rest is a bridge and is kept. The remaining six edges are again the tree of cost 99. The method rests on the cycle property: every edge deleted is the heaviest edge of some cycle.

All four algorithms rest on the same cut and cycle properties. If the weights are distinct, the minimum spanning tree is unique, so all of them find the same tree.

17.5.2 The minimum spanning tree and the shortest path tree

The minimum spanning tree resembles another structure that is often confused with it: the **shortest path** tree. When the shortest paths from a source vertex to every vertex are combined, a tree results. Both are spanning trees, but they minimize different things.

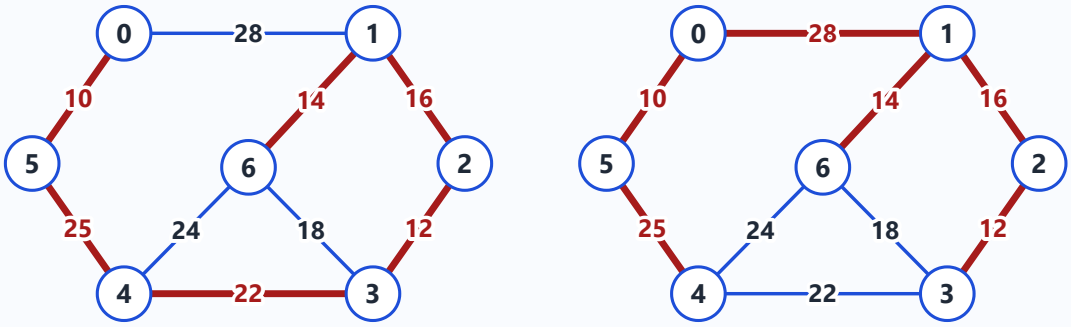


Figure 17.12: The minimum spanning tree of the seven-village graph (left, total 99) and the shortest path tree from 0 (right, total 105).

- **The minimum spanning tree** minimizes the total edge cost. It has no root; it does not matter from which vertex one looks. Going from 0 to 1 in the tree follows the path 0-5-4-3-2-1 and costs *10 + 25 + 22 + 12 + 16 = 85*. The direct edge (0,1), however, is 28.
- **The shortest path tree** minimizes the distance of each vertex from the source one by one. Starting from 0, the distance to 1 is 28, to 2 it is 44, to 3 it is 56, to 4 it is 35, to 5 it is 10 and to 6 it is 42. The edges of the tree are (0,1), (0,5), (1,2), (1,6), (2,3), (5,4), and its total cost is 105; this is larger than 99.

Which tree is needed depends on the question. If the total cost of laying cable to the villages is asked, the minimum spanning tree is sought; if the quickest travel from the center to each village is asked, the shortest path tree is sought. Dijkstra's algorithm, which finds the shortest path tree, is very similar to Prim's: it also grows a single tree and in every round picks the vertex with the smallest key. The only difference is the definition of the key. In Prim's the key is the cost of the single edge connecting the vertex to the tree; in Dijkstra's it is the total length of the path from the source to that vertex.

17.5.3 Kruskal or Prim?

	Kruskal	Prim
Growing structure	forest (disjoint clusters merge)	a single tree (TV)
Selection criterion	the cheapest among all edges	the cheapest edge leaving TV
Cycle check	union-find required	not needed ($v \notin TV$)
Data structure	sorted array or min-heap + sets	`lowcost[]` array or min-heap

	Kruskal	Prim
Time	*O(e log e)*	$O(n^2)$ or *O(e log n)*
Where it is good	sparse graph, edge list	dense graph, cost matrix
Disconnected graph	gives a minimum spanning forest	covers only the component of the start vertex

Table 17.10: Comparison of Kruskal and Prim.

The choice is determined by the density of the graph and the representation at hand. If the edges come as a list from a file and the graph is sparse, Kruskal is the most natural choice: the list is sorted as it is. If the graph is given as a cost matrix and is dense, Prim with arrays is both short and fast. On a sparse graph held with adjacency lists, Prim with a heap and Kruskal give close results. The difference in the last row is also important. Kruskal does not stop on a disconnected graph; it tries all the edges and finds the minimum spanning tree of each component, that is, a forest. Prim stops when it finishes the component of the start vertex; it has to be started again for the other components.

17.5.4 An approach to problem solving

The story of Kruskal and Prim teaches a method as well. An engineer who meets a new problem first asks: How has this problem been solved before? Which data structures have been proposed in the literature? How does the world solve this, and how should I? The instructor also opposes the comfort of “we have mastered algorithms”. These algorithms are still being worked on today. Knowing which structure to use where while solving a problem is the real gain of a data structures course.

The slides express this attitude with a saying about mastery. The saying lists the stages of following the path: looking at the master, following the master, walking with the master, seeing like the master and finally becoming the master. That is, first the solution in the literature is learned and applied; the original solution comes after that. A student who codes Kruskal by keeping lists has skipped the first stage. A student who sees how sets and the heap fit this problem has moved on to the later stages.

17.6 Exam-style question

The coding of Kruskal is given as homework; the following question is of the kind that can be asked in exams. For the graph with six vertices in Figure 17.13, three things are required:

1. Find the minimum spanning tree with Kruskal and write an “add” or “cycle” decision for each edge.
2. Start Prim from A and write the order of the chosen edges.
3. Give the final state of the `parent[]` array of union-find working with weighted union.

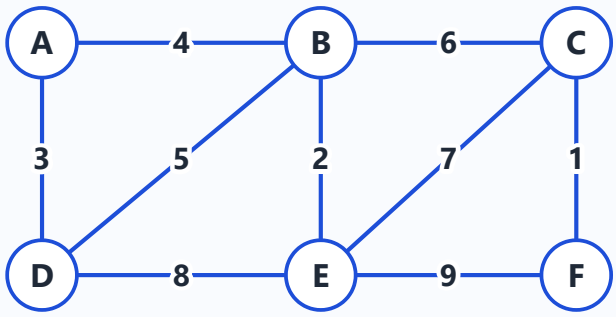


Figure 17.13: The graph of the exam question. Edges: (A,B) 4, (B,C) 6, (A,D) 3, (B,D) 5, (B,E) 2, (C,E) 7, (C,F) 1, (D,E) 8, (E,F) 9.

Kruskal. The edges are sorted: (C,F) 1, (B,E) 2, (A,D) 3, (A,B) 4, (B,D) 5, (B,C) 6, (C,E) 7, (D,E) 8, (E,F) 9. The vertices A..F are numbered 0..5 and, for each edge, `unite(parent, find(parent, u), find(parent, v))` is called. Table 17.11 gives the decisions and the `parent` array.

Edge	w	Roots	Decision	`parent[]` (A B C D E F)
—	—	—	start	`-1 -1 -1 -1 -1 -1`
(C,F)	1	C, F	add	`-1 -1 -2 -1 -1 2`
(B,E)	2	B, E	add	`-1 -2 -2 -1 1 2`
(A,D)	3	A, D	add	`-2 -2 -2 0 1 2`
(A,B)	4	A, B	add	`-4 0 -2 0 1 2`
(B,D)	5	A, A	cycle	unchanged
(B,C)	6	A, C	add	`-6 0 0 0 1 2`

Table 17.11: Kruskal on the exam question and `parent[A..F]`.

When the fifth edge is chosen, $n - 1 = 5$ is reached and the algorithm stops; (C,E), (D,E), (E,F) are not looked at. The tree is {(C,F), (B,E), (A,D), (A,B), (B,C)}, with cost $1 + 2 + 3 + 4 + 6 = 16$. (B,D) forms a cycle, because B and D are already connected through A. For sets of equal size the first parameter stays the root: in (A,B), A and B both have 2 elements, and A stays the root. In (B,C), the set of A has 4 and that of C has 2 elements; C goes under A. The final array is `[-6, 0, 0, 0, 1, 2]`: A is the root of the set of six elements; the parent of E is B and the parent of F is C.

Prim (from A). The edges of A are (A,B) 4 and (A,D) 3: (A,D) is chosen. TV = {A, D}; the candidates are (A,B) 4, (D,B) 5, (D,E) 8: (A,B) is chosen. TV = {A, B, D}; the candidates are (B,E) 2, (B,C) 6, (D,E) 8: (B,E) is chosen. TV = {A, B, D, E}; the candidates are (B,C) 6, (E,C) 7, (E,F) 9: (B,C) is chosen. In the last round (C,F) 1 is chosen. The order is (A,D) 3 → (A,B) 4 → (B,E) 2 → (B,C) 6 → (C,F) 1, with total 16. The tree is the same as Kruskal's; since the weights are distinct, it cannot be otherwise. The noteworthy point is that Prim leaves the cheapest edge (C,F) for last: until C enters the tree, this edge is not a candidate.

All values were verified with a test driver. The graph has 55 spanning trees, and this is the only one with cost 16.

17.7 Homework: coding Kruskal with data structures

For next week the instructor asks everyone to code Kruskal's algorithm as described in class. The expected program uses a **min-heap** and **union-find**. Its input is a list of edges; its output is the edges of the tree and the total cost. The requirements of the homework are:

- The edge structure is of the form `edge {u, v, w}`. The heap is built bottom-up from the edge array ($O(e)$).
- `find` must use path compression and `unite` the weighting rule.
- If the graph is not connected ($|T| < n - 1$), the program must print "no minimum spanning tree".
- On the seven-village example it must produce output showing that it finds 99 and that (3,6) and (4,6) are rejected.

A checklist to look at before submission is also given. One must stop when $t = n - 1$, not take from the heap e times. A `NULL` check must be made after every `malloc` call, and every block allocated must be released with `free` at the end of the program. If there are edges of equal weight, the minimum spanning tree may not be unique; the programs of different students may find different trees, but the total cost is the same in all.

The parts given in this chapter contain all the building blocks of the homework: the `edge` structure and `unite`, `find` with path compression, the sift-down logic (the `heap_pop_min` in Prim with a heap) and the main loop. The real work of the homework is to combine them around a heap that uses the edge weight as the key.

17.8 Complexity summary

Method	Structure	Time	Explanation
Trying all subsets	—	$C(e, n-1) \times O(n)$	exponential; not practical
Kruskal, bad coding	lists + search	$O(e \cdot n)$	correct but not coded this way
Kruskal	sorting or min-heap + union-find	$O(e \log e)$	forest grows, clusters merge
Prim	arrays (`lowcost` , `closest`)	$O(n^2)$	dense graph
Prim	binary heap + adjacency list	$O(e \log n)$	sparse graph
Sollin (Borůvka)	component labels	$O(e \log n)$	at most $\lg n$ rounds; parallel
DFS/BFS spanning tree	stack / queue	$O(n + e)$	unweighted; any spanning tree

Table 17.12: Costs of the methods for spanning trees and minimum spanning trees.

The $O(n)$ factor in the first row is the cycle check of each choice. Even in a small graph like the seven villages, 84 choices are tried; in a graph with 30 vertices and 60 edges, $C(60, 29)$ exceeds 10^{17}. DFS and BFS in the last row find any spanning tree because they do not look at the weights. If all the edges have equal weight, every spanning tree is minimum and is found in linear time.

17.9 Common mistakes

1. **Checking for a cycle in Kruskal's algorithm by asking "have both endpoints been chosen before?".** The two endpoints can be in different clusters. In the seven-village example (1,2) gets caught by this wrong rule, although it is an edge of the MST. The correct check is `find(u) != find(v)`.
2. **Not removing from the candidate set in Prim's algorithm the edges with both endpoints in TV.** (0,1) is a candidate for five rounds, and in the sixth round both its endpoints are in the tree. If this edge is chosen, a cycle forms. In the array version this mistake is forgetting the condition `!in_tree[v]` in the update loop; in the heap version it is not skipping stale records.
3. **Confusing the minimum spanning tree with the shortest path tree.** The MST minimizes the total cost, the shortest path tree minimizes the individual distances. In the seven-village example the distance from 0 to 1 is 85 in the MST and 28 in the shortest path tree; the totals are 99 and 105.
4. **Forgetting the $|T| < n - 1$ check on a disconnected graph.** Kruskal stops silently when the edges run out and is left with a forest. In Prim's, if `cheapest` returning −1 is not checked, `in_tree[-1]` is written; this is a write outside the array.
5. **Uniting non-root vertices in union-find.** `unite` expects two roots. `find` is called first, and then the roots found are united.
6. **Using `INT_MAX` for infinity and getting overflow in sums.** Even if the comparison is safe, the addition overflows and leads to undefined behavior. A constant larger than the largest possible sum but not overflowing is chosen.

17.10 Exercises

1. In the seven-village example, if the cost of the edge (3,6) were 11 instead of 18, what would the

minimum spanning tree and its cost be?

Solution

The sorted edges: (0,5) 10, (3,6) 11, (2,3) 12, (1,6) 14, (1,2) 16, (3,4) 22, (4,6) 24, (4,5) 25, (0,1) 28. Kruskal: add (0,5), add (3,6), add (2,3), add (1,6), (1,2) is a cycle (1 and 2 are connected through 6 and 3), add (3,4), (4,6) is a cycle, add (4,5). The tree is {(0,5), (3,6), (2,3), (1,6), (3,4), (4,5)}, cost *10 + 11 + 12 + 14 + 22 + 25 = 94*. The edge (3,6), now cheaper, has entered the tree, and the most expensive edge on the same cycle, (1,2), has dropped out. In the code output the final `parent` array is `[3, 3, 3, -7, 3, 0, 3]`.

2. Under what condition can Kruskal and Prim give different trees on the same graph?

Solution

Only if there are edges of equal weight. If the weights are distinct, the minimum spanning tree is unique and every correct algorithm finds it. With ties the two algorithms can make different choices among equal edges. For example, in a triangle all of whose edges are 1, Kruskal takes the first two edges in the sorted order. Prim takes two edges at the start vertex, and these may be different. The total cost is still the same.

3. On the eight-vertex example graph (Figure 17.1), write the edges of the DFS and BFS spanning trees starting from 7.

Solution

With sorted adjacency lists, DFS(7): (7,3), (3,1), (1,0), (0,2), (2,5), (2,6), (1,4). Since the neighbors of 5, 2 and 7, are marked, the search moves to 2 and from there to 6; finally it returns to 1 and takes 4. BFS(7): (7,3), (7,4), (7,5), (7,6), (3,1), (5,2), (1,0). Both trees have seven edges.

4. How many spanning trees does K_4 have? How many are paths and how many are stars?

Solution

16 (4^{4-2}). Stars are determined by the choice of the center: 4. Paths are orderings of the four vertices, and reading in reverse gives the same path: *4!/2 = 12*. *12 + 4 = 16*.

5. If the edge weights are integers between 1 and 10, how does the sorting step of Kruskal drop to

O(e)?

Solution

Instead of a comparison-based sort, **counting sort** or bucket sort is used. A bucket is opened for each weight, the edges are distributed into the buckets by weight, and the buckets are read in order. The cost is $O(e + 10) = O(e)$. The total time of Kruskal then comes down to the union-find operations.

6. Why is the $O(n^2)$ version of Prim bad on a sparse graph?

Solution

In every round it scans n vertices, regardless of the number of edges. On a graph with $e \approx n$, the version with a heap takes $O(e \log n) = O(n \log n)$; $O(n^2)$ is much larger than that. On a random graph with 5000 vertices and 20,000 edges, the measured times are 0.106 and 0.003 seconds.

17.11 Summary

- A spanning tree is the subgraph of a connected graph that connects all its vertices with $n - 1$ edges and no cycle. DFS and BFS each produce a spanning tree; they split the edges into tree edges and nontree edges.
- Of the properties connectivity, acyclicity and $n - 1$ edges, any two imply the third. Adding an edge to a tree forms a single cycle, and deleting an edge from a tree splits it in two.
- The number of spanning trees grows quickly: n^{n-2} in K_n. Trying them all is impractical; testing isomorphism is also a hard problem in its own right.
- A minimum spanning tree is a spanning tree with the smallest total weight. The greedy choice gives the right result; the justification is the cut property. If the weights are distinct, the tree is unique.
- Kruskal tries edges in increasing order of cost, catches cycles with union-find and stops when $n - 1$ edges have been chosen: $O(e \log e)$. The intermediate states are forests.
- Union-find is a `parent` array; at a root it holds the negative of the set size. The weighting rule keeps the height at $O(\log n)$, and path compression brings the cost per operation down to a constant in practice.
- Prim grows a single tree by the cheapest edge and needs no cycle check. It takes $O(n^2)$ with the `lowcost` / `closest` arrays and $O(e \log n)$ with adjacency lists and a heap.
- Same idea, different structure: Kruskal, which takes $O(e \cdot n)$ with list search, drops to $O(e \log e)$ with sets.
- Kruskal or Prim with a heap is chosen on a sparse graph, and Prim with arrays on a dense one. Sollin (Borůvka) works in rounds and is suited to parallelization.

- The minimum spanning tree minimizes the total cost, the shortest path tree minimizes the distances from the source; the two are different trees.

17.12 Key terms

spanning tree minimum spanning tree forest isomorphic complete graph weighted graph
connected component cycle greedy method Kruskal's algorithm Prim's algorithm
Sollin's algorithm disjoint sets find union weighting rule path compression
inverse Ackermann function min-heap priority queue bottom-up heap construction adjacency list
depth-first search breadth-first search shortest path counting sort

---

17.13 Test Yourself

1. How many edges does every spanning tree of a connected graph with 10 vertices and 15 edges have? How many edges does DFS leave out of the tree on this graph?

   Answer

   A spanning tree has $n - 1 = 9$ edges. The remaining $15 - 9 = 6$ edges are nontree edges; this number is independent of the type of traversal and of the start vertex.

2. How many spanning trees does K_5 have? How many of them are stars?

   Answer

   By Cayley's formula, $5^3 = 125$. The center of the star can be chosen in five ways: 5 stars.

3. On the seven-village graph, which edges does Kruskal discard because they form cycles? Which edge does it never look at?

   Answer

   (3,6) 18 and (4,6) 24 form cycles: both have endpoints in the same cluster at that moment. (0,1) 28 is never looked at, because when (4,5) 25 is chosen the six edges are complete.

4. `parent` = `[-3, 0, 0, -2, 3]` is given. What does `find(parent, 2)` return? If `unite(parent,`

`0, 3)` is then called, what does the array become?

Answer

`parent[2] = 0`, `parent[0] = -3 < 0`: the root is 0. In the union, since `parent[0] = -3 <= parent[3] = -2`, 0 stays the root: `parent[0] = -5`, `parent[3] = 0`. The array becomes `[-5, 0, 0, 0, 3]`; the parent of 4 is still 3.

5. If Prim is started from 3 on the seven-village graph, in what order are the edges chosen? Does the cost change?

Answer

(3,2) 12, (2,1) 16, (1,6) 14, (3,4) 22, (4,5) 25, (5,0) 10. The cost is again 99; since the weights are distinct the tree is unique, and only the order of selection changes.

6. If the line `if (in_tree[u]) continue;` is deleted in Prim with a heap, what happens on the seven-village graph?

Answer

Stale records are processed too. When the records 6:18, 6:24 and 1:28 come out, 6 and 1 are "taken into the tree" again and their neighbors are scanned again. Since all the neighbors are marked, `lowcost` and `closest` are not corrupted. But if the number of vertices entering the tree or the total is counted in this line, vertices are counted more than once; line 5 guarantees that each vertex is processed exactly once.

7. A road network with 1000 vertices and 3000 edges and a complete graph with 1000 vertices and about 500,000 edges are given. Which Prim implementation should be chosen for each?

Answer

The road network is sparse: Prim with a heap takes about $e \log n \approx 3000 \cdot 10 = 30{,}000$ steps and Prim with arrays $n^2 = 10^6$; the heap is chosen. The complete graph is dense: the heap version does about $5 \cdot 10^5 \cdot 10 = 5 \cdot 10^6$ heap work, while the array version stays at 10^6; Prim with arrays is chosen.

8. In a graph whose edge weights are all distinct, can the heaviest edge be in the minimum spanning tree?

Answer

It can, but only if that edge is not on any cycle, that is, if deleting it disconnects the graph. For example, if a vertex is connected by a single edge, that edge is in every spanning tree however expensive it is. If the edge is on a cycle, by the cycle property it cannot be in the tree.

17.14 Further reading

- E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2nd ed., Section 6.3. The Kruskal and Prim pseudocode and the seven-village example in this chapter follow its presentation; Sollin's algorithm is also described in the same section.
- T. H. Cormen et al., *Introduction to Algorithms*, 4th ed., Chapters 19 and 21. The amortized analysis of disjoint-set forests, the proof of the inverse Ackermann bound and the general proof of the cut property under the name “safe edge” are there.
- R. Sedgewick, *Algorithms in C, Part 5: Graph Algorithms*, 3rd ed., Chapter 20. C implementations of Prim, Kruskal and Borůvka are given together with experimental comparisons on large random graphs.

17.15 References

1. E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2nd ed., Silicon Press, 2008. Section 6.3, Minimum Cost Spanning Trees.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022. Chapter 19, Data Structures for Disjoint Sets; Chapter 21, Minimum Spanning Trees.
3. R. Sedgewick, *Algorithms in C, Part 5: Graph Algorithms*, 3rd ed., Addison-Wesley, 2002. Chapter 20, Minimum Spanning Trees.

PART IV · CHAPTER 18

Shortest Paths and Activity Networks

This chapter takes up two big questions about weighted directed graphs: what is the shortest path from one vertex to another, and when can a project made of interdependent tasks finish at the earliest? The first question starts with brute force, and counting shows why that solution is unusable. Dijkstra's greedy algorithm is then built in C, traced, and analyzed; the path itself is read from a single predecessor array. Bellman-Ford handles negative edges, and Floyd-Warshall handles all pairs of vertices. The second half is devoted to activity networks. In activity-on-vertex networks, a topological order is found in linear time with counted headers and an embedded stack. In activity-on-edge networks, the same routine computes the earliest times, and its twin running on the reversed graph computes the latest times. Together the two arrays give the slack of every activity and the critical path of the project. Each routine comes with a line-by-line walkthrough, a trace table, and a complexity analysis.

LEARNING OUTCOMES

Model
Cast problems such as a railway network, network latency, or a project plan as weighted directed graphs.

Implement
Trace Dijkstra's algorithm as $O(n^2)$ C code, justify its correctness, and print the path from the predecessor array.

Extend
Apply Bellman-Ford to graphs with negative edges and Floyd-Warshall to all pairs; decide which algorithm fits which problem.

Order
Find a topological order of an activity-on-vertex network in $O(n + e)$ time with counted headers and an embedded stack, and detect cycles.

Compute
Compute the earliest and latest start times of an activity-on-edge network step by step with forward and backward passes.

Interpret
Find the slacks, identify the critical activities and the critical path, and explain how a change in one duration affects the project.

18.1 The shortest-path problem

Picture a map of cities and the railway lines between them. Every line has a length, and the traveler's question is simple: what is the shortest route from one city to another? The same question arises in a computer network. There the vertices are routers, and the edges are links between two routers. The value of an edge is no longer a length but the time a packet waits on that link, which grows as traffic gets heavier. A router deciding which neighbor to forward a packet to is, in effect, solving the shortest-path problem again every second.

Both examples share one model, the **weighted directed graph**. Vertices stand for cities or routers, and edges stand for connections. Every edge has a cost (a weight). Making edges directed is realistic: a one-way road, or a link that is congested in only one direction, gives different costs in the two directions. The length of a path is the sum of the costs of its edges. The number of edges does not matter here; a path of three edges can be shorter than a path of one.

Definition 18.1

Let $G = (V, E)$ be a weighted directed graph and let $cost(\langle u, w \rangle)$ be the weight of an edge. The **length** of a path $v_0, v_1, \ldots, v_k$ is the sum $cost(\langle v_0, v_1 \rangle) + \ldots + cost(\langle v_{k-1}, v_k \rangle)$. Among the paths from x to y, one of minimum length is called a **shortest path**.

The problem contains two separate questions. First, is there any path from x to y at all? This is a reachability question, answered by a depth-first or breadth-first traversal of the graph. Second, if there are several paths, which one is shortest? This chapter is about the second question. The problem can be posed in several forms. In the **single-source single-destination** version, a start and a target vertex are given. In the **single-source all-destinations** version, the shortest paths from one source vertex to every other vertex are wanted. In the **all-pairs** version, a separate answer is sought for every pair of vertices.

The shortest-path problem should not be confused with a similarly named one. Finding the shortest tour that visits every vertex once and returns to the start is the traveling salesman problem. In the traveling salesman problem the order of all vertices is chosen; in the shortest-path problem only the route between two vertices is sought. All known exact algorithms for the former take exponential time, while the latter is solved very efficiently.

Throughout this chapter we use the six-vertex graph of Figure 18.1. It is the example from the textbook by Horowitz and Sahni. The edges leaving 0 go to 1 (50), 2 (10), and 4 (45). There is an edge from 2 to 3 of cost 15 and one from 3 to 1 of cost 20. Vertex 5 has a single outgoing edge (5 → 3); no

vertex has an edge into 5.

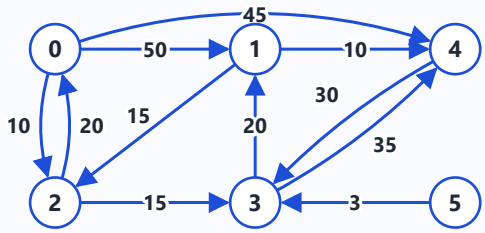


Figure 18.1: The main example of the chapter: a weighted directed graph with six vertices. Each arrow is an edge; the number on the arrow is the cost of the edge.

Taking 0 as the source, the lengths of all shortest paths appear in Table 18.1. Some of the results are not obvious at a glance. There is a direct edge of cost 50 from 0 to 1, but it is not the shortest path: the path $0 \to 2 \to 3 \to 1$ has length $10 + 15 + 20 = 45$. Three edges are cheaper than one. From 0 to 4, on the other hand, the direct edge (45) is best. The path $0 \to 2 \to 3 \to 4$ costs $10 + 15 + 35 = 60$. No edge leads to 5, so there is no path from 0 to 5.

Target	Shortest path	Length
2	$0 \to 2$	10
3	$0 \to 2 \to 3$	25
1	$0 \to 2 \to 3 \to 1$	45
4	$0 \to 4$	45
5	—	no path

Table 18.1: Shortest paths from 0 to all vertices in the example graph.

The rows of the table are sorted by length. This ordering is no accident; the most efficient algorithm finds the targets in exactly this order. There is also a link between the rows. The shortest path to 1 passes through the shortest path to 3, and that of 3 through the shortest path to 2. Every shortest path is obtained by appending a single edge to a shorter shortest path. These two observations underlie the first half of the chapter.

18.2 The brute-force solution

The first solution that comes to mind for a problem is often **brute force**: try every candidate and pick the best. For shortest paths, the candidates are all simple paths from the source to the target. A simple path is one that never passes through the same vertex twice. When no weight is negative, it is easy to see that a shortest path is simple: if a path visits the same vertex twice, the cycle in between can be cut out without increasing the length. So it suffices to try only simple paths.

For the single-source single-destination version the algorithm goes as follows. Start at the source, move to every unvisited neighbor, and when the target is reached, compare the length of that path with the best value so far. This is a depth-first search done with backtracking. The logic is correct, and so is the result. The problem is the number of paths to try.

The worst case is the **complete graph**, in which every vertex is connected to every other. The complete graph on n vertices is written K_n. Figure 18.2 shows K_4 and K_5. In K_4 there are five paths from 0 to 3: 0-3, 0-1-3, 0-2-3, 0-1-2-3, and 0-2-1-3. In K_5, the paths from 0 to 4 are counted by the number of intermediate vertices. There is 1 path with no intermediate vertex (0-4). There are 3 paths with one intermediate vertex, which can be 1, 2, or 3. For paths with two intermediate vertices, the first can be chosen in 3 ways and the second in 2: 6 paths. For paths with three intermediate vertices, all orderings of 1, 2, and 3 are tried: $3! = 6$ paths. The total is $1 + 3 + 6 + 6 = 16$ paths.

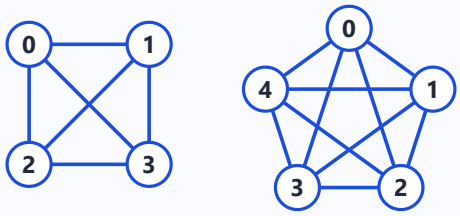


Figure 18.2: The complete graphs K_4 and K_5. The edges of K_4 can be drawn in the plane without crossing; those of K_5 cannot. Planarity does not matter for the number of paths; what matters is that every pair of vertices is connected.

The same count works for general n. Apart from the source and the target there are $n - 2$ vertices. A path with k intermediate vertices is an ordered selection of k of them. The number of ordered selections is $(n - 2)(n - 3)\cdots(n - 1 - k)$. Adding up all values of k gives the number of paths.

Theorem 18.1 (Number of paths in a complete graph)

The number of simple paths between two distinct vertices of K_n is

$$Y(n) = 1 + (n - 2) + (n - 2)(n - 3) + \cdots + (n - 2)!$$

The last term of the sum is the largest, and $Y(n) < e \cdot (n - 2)!$ holds ($e \approx 2.718$).

Proof. The first equality is the count above. For the upper bound, write the sum from the end backward: $(n - 2)! \cdot (1 + 1 + 1/2! + 1/3! + \cdots)$. The series in parentheses is part of the series $1/0! + 1/1! + 1/2! + \ldots$, whose sum is e. ■

The formula was also checked with a program. The program counts, one by one with backtracking, the paths from 0 to $n - 1$ in a complete graph. The results are shown next to the formula in Table 18.2; in every row the two values agree. The last column shows how many times the search advanced along

an edge. This number is about twice the number of paths, so the amount of work grows at the same rate as the number of paths.

n	Number of paths	Formula	Edge steps
4	5	5	9
5	16	16	31
6	65	65	129
8	1,957	1,957	3,913
10	109,601	109,601	219,201
11	986,410	986,410	1,972,819

Table 18.2: Number of simple paths between two vertices of a complete graph (counted by the program and computed from the formula).

The growth is factorial. With 11 vertices there are nearly a million paths; with 20 vertices the number reaches the order of $18! \approx 6 \cdot 10^{15}$. Even a computer trying a billion paths per second would need about two months to finish them. And this is for a single target.

The $O(n^n)$ notation used in the slides is a crude but correct upper bound on this growth. Since $(n-2)! \le n^n$, we can write $Y(n) = O(n^n)$. The tighter bound is $O((n-2)!)$; both are exponential and say the same thing in practice: this method is usable only on graphs of a few vertices. The logic of brute force is correct. But a method this expensive even for a single target does not solve the problem. For something better, we must exploit the structure in the paths.

18.3 Dijkstra's algorithm

18.3.1 The greedy idea

Brute force builds every path from scratch. Yet the observations in Table 18.1 point to something smarter: shortest paths can be found from short to long, and every new path is an earlier path extended by one edge. In 1959 Edsger W. Dijkstra turned this idea into a very efficient algorithm. The algorithm is **greedy**: at each step it makes the choice that looks best at that moment and never revises it.

The algorithm maintains a set S. S consists of the vertices whose shortest paths are final; at the start it holds only the source, 0. For every vertex w outside S, a value $dist[w]$ is also kept.

Definition 18.2

For $w \notin S$, $dist[w]$ is the length of the shortest path that starts at the source, ends at w, and has all its intermediate vertices in S. If no such path exists, $dist[w] = \infty$.

The restriction in the definition matters: *dist[w]* is the shortest among the paths that pass only through *S*, not among all paths. Since *S = {0}* at the start, these paths consist of a single edge, and *dist[w]* equals the cost of the edge from 0 to *w*. The algorithm rests on three observations.

1. The shortest path to the next vertex *u* to be finalized passes, apart from *u* itself, only through vertices in *S*. If some *x* outside *S* lay in the middle of the path, the part from the source to *x* would be shorter, and *x* would have been chosen earlier.
2. Therefore the next vertex is the one outside *S* with the smallest *dist* value. If there is a tie, any of them may be chosen.
3. Once *u* is chosen and joins *S*, new paths through *u* become available. For a vertex *w* outside *S*, if *dist[u] + cost(<u, w>) < dist[w]*, then *dist[w]* is updated with this smaller value. In the new path there is no intermediate vertex between *u* and *w*; after *u* the path goes directly to *w*.

The operation in the third observation is called **relaxation**. The name comes from the fact that *dist[w]* is an upper bound: the value is never smaller than the true shortest distance. Relaxation pulls this bound down, that is, it loosens an overly "tight" estimate. Figure 18.3 shows one relaxation step: for the neighbors w_1, w_2, w_3 of the newly added *u*, new paths through *u* are tried.

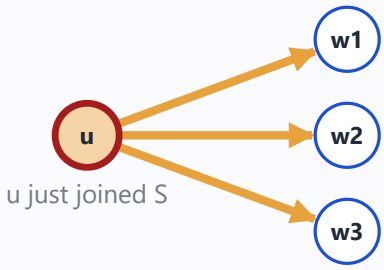


Figure 18.3: The vertex *u* that has just joined *S* opens new paths through itself to its neighbors. For each w_i, the value *dist[u] + cost(<u, w_i>)* is compared with the old one.

The justification of the first observation depends on one condition: edge weights must not be negative. Without it the algorithm goes wrong. To state the correctness precisely, we can write it as a theorem.

Theorem 18.2 (Correctness of Dijkstra's algorithm)

If all edge weights are non-negative, then for the vertex *u* outside *S* with the smallest *dist* value, *dist[u]* is the length of a shortest path from the source to *u*.

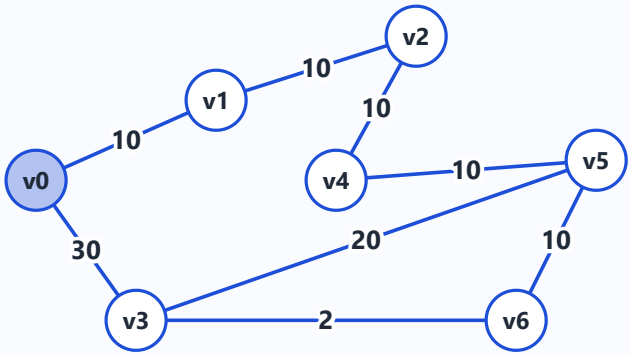
Proof. Take any path *P* from the source to *u*. *P* starts at the source, which is inside *S*, and ends at *u*, which is outside *S*. So *P* has a first vertex *y* outside *S*. The part of *P* from the source to *y* passes only through *S*; by definition its length is at least *dist[y]*. Because *u* was chosen as the vertex with the

smallest value, *dist[y] ≥ dist[u]*. The remaining part of *P*, from *y* to *u*, has length 0 or more because the weights are non-negative. In total, the length of *P* cannot be smaller than *dist[u]*. Since *dist[u]* is itself the length of an actual path, it is the length of a shortest path. ■

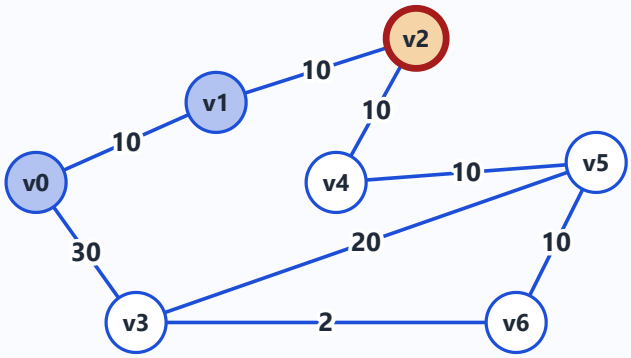
The only weak link in the proof is the sentence "the remaining part is 0 or more." A negative edge breaks it: if one descends from *y* to *u* along a negative edge, *P* can be shorter than *dist[u]*. A concrete example of this situation, and its remedy, appear in the section on negative edges.

18.3.2 A step-by-step example

To see the algorithm at work, take a small graph with seven vertices. Its edges are *v0* → *v1* (10), *v1* → *v2* (10), *v2* → *v4* (10), *v4* → *v5* (10), *v0* → *v3* (30), *v3* → *v5* (20), *v3* → *v6* (2), and *v6* → *v5* (10). The source is *v0*. Figure 18.4 shows four of the seven steps, and Table 18.3 shows all of them. In the figures, dark vertices are in *S*; the table on the right shows the `found` flag (T) and the `dist` value.



	found	dist
v0	T	0
v1	F	10
v2	F	∞
v3	F	30
v4	F	∞
v5	F	∞
v6	F	∞



	found	dist
v0	T	0
v1	T	10
v2	T	20
v3	F	30
v4	F	30
v5	F	∞
v6	F	∞

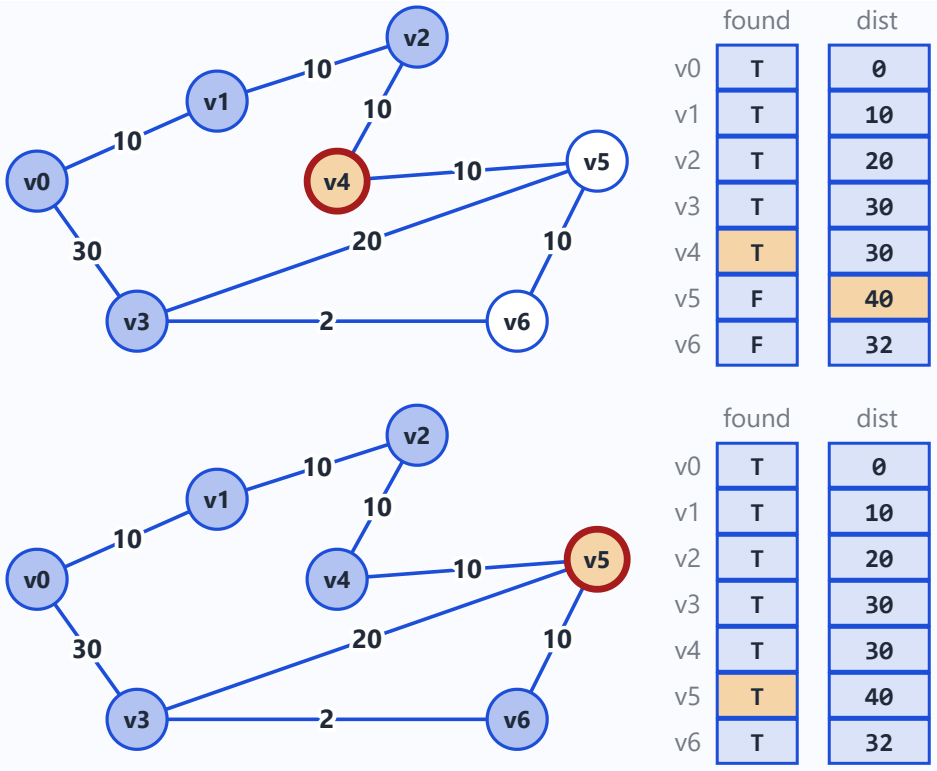


Figure 18.4: Four moments of Dijkstra's algorithm on the seven-vertex graph. Top left, the start: only *v0* is final, and its neighbors take their direct edge costs. Top right, *v2* has joined and *v4* has dropped to 30. Bottom left, *v4* has joined and *v5* has dropped from 50 to 40. Bottom right, all vertices are final.

Step	Joined	v1	v2	v3	v4	v5	v6	Updated
0	v0	10	∞	30	∞	∞	∞	v1, v3 (direct edge)
1	v1 (10)	**10**	20	30	∞	∞	∞	v2 = 20
2	v2 (20)	**10**	**20**	30	30	∞	∞	v4 = 30
3	v3 (30)	**10**	**20**	**30**	30	50	32	v5 = 50, v6 = 32
4	v4 (30)	**10**	**20**	**30**	**30**	40	32	v5 = 40
5	v6 (32)	**10**	**20**	**30**	**30**	40	**32**	none (32 + 10 = 42 > 40)
6	v5 (40)	**10**	**20**	**30**	**30**	**40**	**32**	none

Table 18.3: Order of joining *S* and the changes in distances on the seven-vertex graph (∞: no path yet).

Two rows of the table are especially instructive. In the fourth step the value of *v5* drops from 50 to 40. The first path found to *v5* is *v0* → *v3* → *v5* (30 + 20). When *v4* joins, the four-edge path *v0* → *v1* → *v2* → *v4* → *v5* (10 + 10 + 10 + 10) is found, and it is shorter. A vertex's value may change several times until the vertex enters *S*; it does not count as final before that.

In the fifth step, *v6* is chosen before *v5*. The number of *v6* is larger, but its distance (32) is smaller than that of *v5* (40). The order of selection is determined by distance, not by vertex number. In the third and fourth steps, *v3* and *v4* have equal distances (30). In theory it does not matter which is

chosen first. Because the C code picks the lower-numbered vertex on a tie, *v3* joins first.

18.3.3 The C implementation: data and `min_vertex`

The C version of the algorithm stores the graph as a **cost matrix**. `cost[i][j]` is the cost of the edge <*i, j*>. Where there is no edge, a large constant is written in place of ∞; the diagonal entries are 0. Finalized vertices are marked in the `found` array; this is the set *S*. The `dist` array holds the values from the definition. The code below gives the matrix of the example graph and the routine that finds the unmarked vertex with the smallest distance.

```
#include <limits.h>
#define MAX_VERTICES 6
#define INF 1000      /* no edge */
int cost[][MAX_VERTICES] = {
    {   0,   50,  10, 1000,   45, 1000},
    {1000,    0,  15, 1000,   10, 1000},
    {  20, 1000,   0,   15, 1000, 1000},
    {1000,   20, 1000,   0,   35, 1000},
    {1000, 1000, 1000,  30,    0, 1000},   /* missing row */
    {1000, 1000, 1000,   3, 1000,    0}};  /* 100 -> 1000 */
int min_vertex(int dist[], int n, short found[])
{
    int i, min = INT_MAX, pos = -1;
    for (i = 0; i < n; i++)
        if (dist[i] < min && !found[i]) {
            min = dist[i];  pos = i;
        }
    return pos;
}
```

- **Line 1.** The constant `INT_MAX` is in the header `limits.h`; it is the largest value an `int` can hold. If the header is not included, the compiler does not recognize `INT_MAX`.
- **Lines 2–3.** The number of vertices and the constant that represents infinity. 1000 is chosen for `INF`; no real path in the example is that long.
- **Lines 4–10.** The cost matrix. The row gives the vertex the edge leaves, and the column the vertex it enters. The values `50, 10, 1000, 45` in row 0 are the edges 0 → 1, 0 → 2, 0 → 3 (no edge), and 0 → 4. The first dimension is left empty; the compiler infers the number of rows from the initializer list. The second dimension must be written, because the compiler has to know how many `int`s make up a row.
- **Lines 11–19.** `min_vertex`. `min` starts at the largest `int` and `pos` at an invalid index (−1). The loop looks, among the unmarked vertices, for the one with the smallest `dist` value. The comparison is strict less-than (`<`); among equal values the first one seen, that is, the lower-numbered vertex, is kept.

Two rows of the matrix carry comments. In the textbook's matrix, row 4 is missing; it was completed by reading the graph (the only edge from 4 is 4 → 3, cost 30). In row 5, too, the book writes 100 for infinity; it was changed to 1000 to be consistent with the other rows. Small typos in textbooks come to light the moment you actually run the code. That is why it matters that the code traced on paper and the code that runs are the same.

The choice of the value `INF` is subtler than it looks. The first idea is `INT_MAX`, but that value causes overflow in relaxation: if either term of the sum `dist[u] + cost[u][w]` is `INT_MAX`, the result does not fit in an `int`. In C, signed integer overflow is undefined behavior; in practice the value often wraps to a negative number, and the code believes "a shorter path was found." With a constant such as 1000 there is no such danger: `INF + INF` is 2000 and fits easily. There is a price: the constant must be chosen larger than the longest real path in the graph. Otherwise a real path is confused with "no path."

The only case in which the routine returns −1 is when no unmarked vertex remains. If an unreachable vertex remains, the routine may still select it, because its value is 1000, which is less than `INT_MAX`. This is harmless. The value of an unreachable vertex is 1000; relaxations through it never produce a value below 1000, because the weights are non-negative. Still, the calling routine should check for a return of −1; an unchecked access `found[-1]` writes outside the array.

> **Common mistake**
>
> Writing the condition as `!found(w)`. Round parentheses call the array as if it were a function, and the compiler reports an error such as "called object is not a function." An array element is accessed with square brackets: `!found[w]`. Another similar mistake is using `FALSE` and `TRUE` without defining them; they are not C keywords and must be defined with `#define`.

18.3.4 `shortest_path`: the main routine

The main routine takes the source vertex, the cost matrix, and three arrays. `dist` and `found` are used with the meanings above. The `pred` array is kept to recover the path itself; the details are in the next section.

```
void shortest_path(int v, int cost[][MAX_VERTICES],
                   int dist[], int pred[],
                   int n, short found[])
{
    int i, u, w;
    for (i = 0; i < n; i++) {                  /* A1 */
        found[i] = FALSE;
        dist[i] = cost[v][i];
        pred[i] = v;              /* kept for the path */
    }
    found[v] = TRUE;  dist[v] = 0;
    for (i = 2; i < n; i++) {         /* n-2 rounds */
        u = min_vertex(dist, n, found);
        if (u < 0) break;             /* none left */
        found[u] = TRUE;
        for (w = 0; w < n; w++)
            if (!found[w] &&
                dist[u] + cost[u][w] < dist[w]) {
                dist[w] = dist[u] + cost[u][w];
                pred[w] = u;
            }
    }
}
```

The routine has four parts, and the analysis uses their names:

- **A1 (lines 6–10).** Initial values. Every vertex is unmarked. `dist[i]` is the cost of the direct edge from the source to `i`; this is the value the definition gives for $S = \{v\}$. With the assignment `pred[i] = v`, every vertex initially counts the source as its predecessor.
- **Line 11.** The source is marked and its distance set to 0. The diagonal of the matrix is already 0; the assignment is written out anyway, because code should not rely on the diagonal.
- **A2 (line 12).** The outer loop runs from $i = 2$ to $n - 1$, that is, $n - 2$ rounds. The source is already final. Each round finalizes one more vertex. After $n - 2$ rounds a single unmarked vertex remains. For that vertex all other vertices are in S, so the value from the definition is the length of the shortest path. One more round would only mark it; no unmarked vertex would be left to update, so nothing would change.
- **A3 (lines 13–15).** The unmarked vertex `u` with the smallest distance is found and marked. If −1 is returned, no unmarked vertex is left, and the loop exits.
- **A4 (lines 16–21).** Relaxation. For every unmarked `w`, the path through `u` is tried. If it is shorter, both the distance and the predecessor are updated.

The condition in lines 17–18 has two parts, and their order matters. The `&&` operator is evaluated left to right with short-circuiting: if `found[w]` is true, the addition is never performed. This keeps the values of finalized vertices from being corrupted. A shorter path to a finalized vertex cannot be found anyway (Theorem 18.2), but having the code check this explicitly improves both readability and safety.

The cost-matrix parameter is written as `int cost[][MAX_VERTICES]`. C does not copy an array into a function; it passes the address of the first element. In a two-dimensional array, computing the address of `cost[u][w]` requires the row length: the address is the base plus *(u · MAX_VERTICES + w) · sizeof(int)*. That is why the second dimension must be written in the parameter. For the same reason, assignments to the arrays `dist`, `pred`, and `found` inside the routine change the caller's arrays; this is how the results come back.

18.3.5 Code trace

The call `shortest_path(0, cost, …)` is traced on the example graph. Table 18.4 gives, for every round, the selected vertex, the relaxations performed, and the `dist` array at the end of the round. The values come from the output the program printed at the end of each round.

Round (`i`)	Selected `u`	Relaxations	`dist[0..5]`
A1	—	row 0 is copied	0 50 10 1000 45 1000
2	2 (10)	`w` = 3: 10 + 15 = 25 < 1000 → `pred[3] = 2`	0 50 10 25 45 1000
3	3 (25)	`w` = 1: 25 + 20 = 45 < 50 → `pred[1] = 3`	0 45 10 25 45 1000
4	1 (45)	none (1 → 4: 45 + 10 = 55 > 45)	0 45 10 25 45 1000
5	4 (45)	none (the only edge of 4 goes to 3, which is marked)	0 45 10 25 45 1000

Table 18.4: Trace of `shortest_path(0, …)` on the example graph (1000: infinity).

In the third round a path to 4 through 3 is also tried: 25 + 35 = 60. Since this is greater than 45, the value does not change. In the fourth round the distances of 1 and 4 are equal (45). Because `min_vertex` uses a strict less-than comparison, it picks 1, the first one it sees. The loop runs *n − 2 = 4* rounds and ends. Vertex 5 was never selected; its value stays 1000 and is read as "no path." Figure 18.5 shows the first and last moments of the trace.

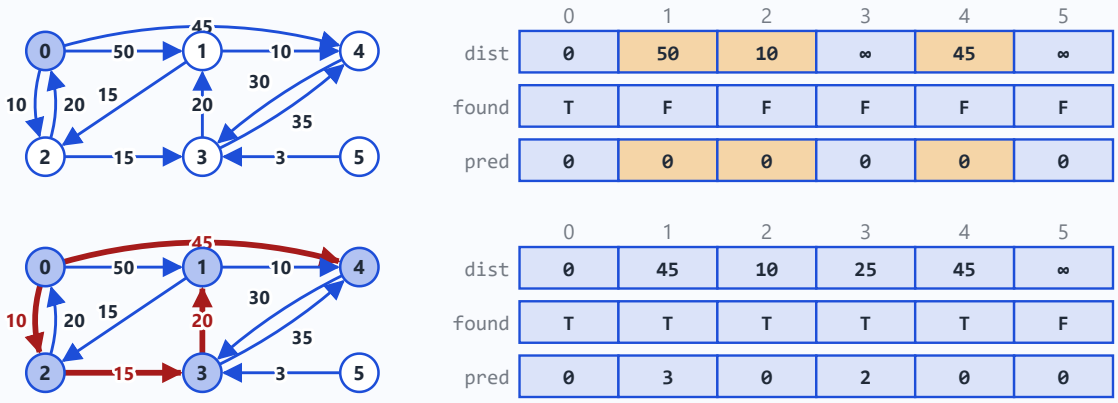


Figure 18.5: The first and last moments of the trace of `shortest_path(0, …)`. Top, after A1: the `dist` array is a copy of row 0, and all elements of `pred` are 0. Bottom, after four rounds: the bold edges form the shortest-path tree.

The result is identical to Table 18.1: 2 → 10, 3 → 25, 1 → 45, 4 → 45, and no path to 5. The order of selection is also the order of length. If 2 is taken as the source, the distances are 20 to 0, 15 to 3, 35 to 1 (through 3), and 45 to 4 (through 3 and 1); again there is no path to 5. These values were also found by running the program.

18.3.6 Analysis and better data structures

The cost is computed separately for each part. A1 runs n rounds once: $O(n)$. The outer loop of A2 runs $n - 2$ rounds. In each round A3, that is `min_vertex`, scans the whole array: $O(n)$. A4 also tries every `w` once: $O(n)$. The total is

$$n + (n - 2) \cdot (n + n) \approx n + 2n^2 = O(n^2)$$

This bound also holds in the best case: whatever the number of edges, every round makes two full scans. So the running time of the algorithm is $\Theta(n^2)$. The extra memory is three arrays: $O(n)$. Compared with the factorial growth of brute force, the difference is striking. With 20 vertices brute force takes months, while this routine finishes after a few hundred operations. Even finding the largest element of the matrix requires looking at n^2 entries; Dijkstra finds all the shortest paths in time of the same order.

$O(n^2)$ is good for dense graphs. For sparse graphs there are two bottlenecks. The first is A3: to find the smallest element, the whole array is scanned in every round. Yet returning the smallest element quickly is exactly the job of a **priority queue** implemented as a **min-heap**. If the distances are kept in a min-heap, the smallest element is removed in $O(\log n)$. The second is A4: because a matrix is used, n columns are scanned in every round, whereas in reality only the neighbors of `u` are of interest. With an adjacency list, A4 takes time proportional to the out-degree of `u`.

When both improvements are applied together, the total cost is easily miscalculated. A common mistake is to assume the result is $O(n \log n)$. The correct calculation is as follows. Every vertex is removed from the heap once: n times $O(\log n)$. Every edge is relaxed once, and each successful relaxation decreases a value in the heap, which takes $O(\log n)$. Since the number of edges is e, the total is

$$O(n \log n + e \log n) = O((n + e) \log n)$$

A more complex structure called a Fibonacci heap brings the decrease-key operation down to $O(1)$ amortized and improves the bound to $O(e + n \log n)$. Table 18.5 puts the three approaches side by side.

Method	Data structure	Time	Suited to
Brute force	all paths	$O(n^n)$	a few vertices
Dijkstra	matrix + arrays	$O(n^2)$	dense ($e \approx n^2$)

Method	Data structure	Time	Suited to
Dijkstra	min-heap + adjacency list	$O((n + e) \log n)$	sparse ($e \approx n$)

Table 18.5: Three approaches to single-source shortest paths.

Which version is better depends on the graph. In a sparse graph, such as a road map, every vertex has a few neighbors and e is of the order of n. Here the heap version drops to $O(n \log n)$ and beats the array version by a clear margin. In a dense graph, however, $e \approx n^2$ and the heap version rises to $O(n^2 \log n)$; the simple array version is faster. Once you have learned the priority queue, searching an array for the minimum may look clumsy. But the right question is not "which structure is more elegant?" but "which operation is performed how many times on this graph?"

18.4 Recovering the path: the predecessor array

18.4.1 Why one array is enough

`shortest_path` finds the lengths, but a traveler is not content with a length alone; the traveler also wants to know which cities the route passes through. Once the cost is known, the path is next. The first idea is to keep, for every vertex, all the vertices of its path in a separate list. This wastes memory and requires copying the list at every update. Building the path forward from the source is not possible either: when leaving the source, it is not known which edge to take until the algorithm has finished.

The solution is a single array. The shortest path to each vertex is obtained by appending a single edge to the shortest path of its **predecessor**. So it suffices to store, for every vertex, only the previous vertex on its path.

> **Definition 18.3**
>
> On the shortest path found from the source to t, the vertex immediately before t is called the **predecessor** of t and is stored in `pred[t]`. The array that holds the predecessors of all vertices is called the **predecessor array**.

A single array suffices because of a property of shortest paths. If a shortest path from the source to t passes through u, then the part of that path from the source to u is also a shortest path from the source to u. If the part could be replaced by a shorter one, the whole path would become shorter too. This property is called **optimal substructure**. As a consequence, all shortest paths form a tree. Its root is the source, and the parent of each vertex is its predecessor. This tree is called the **shortest-path tree**. Like the parent array used for sets, the predecessor array stores, for each vertex of this tree, the link to its parent.

The array is updated during relaxation. Line 20 of `shortest_path` performs the assignment `pred[w] = u` when a relaxation succeeds: the best path to `w` now passes through `u`. As the cost changes, the predecessor changes with it. In the example, while the value of 1 dropped from 50 to 45, its predecessor moved from 0 to 3. When the algorithm ends, the predecessor array of the example graph is `pred = {0, 3, 0, 2, 0, 0}`. The predecessor of 1 is 3, that of 3 is 2, and that of 2 is 0.

The array is read backward, not forward. Starting from the target and walking along predecessors, one arrives at the source: 1 ← 3 ← 2 ← 0. This order is the reverse of the path. To write it in the right order, either recursion or a stack is used. Both rely on the same principle: last in, first out.

18.4.2 Recursive path printing

The recursive solution is short and elegant:

```
void print_path(int pred[], int s, int t)
{
    if (t == s) {
        printf("%d", s);
        return;
    }
    print_path(pred, s, pred[t]);   /* predecessor first */
    printf(" -> %d", t);
}
```

- **Lines 3–6.** Base case: if the target is the source itself, the path is a single vertex; the source is printed and the routine returns.
- **Line 7.** First the path up to the predecessor is printed. The recursive call moves one step closer to the source by replacing `t` with its predecessor.
- **Line 8.** After the path to the predecessor has been printed, `t` is printed. Because the printing is done *after* the call, the vertices come out during the return, that is, from the source toward the target.

The routine works correctly only if the predecessor chain ends at the source. This holds for every reachable vertex, because the predecessors are the parent links of the shortest-path tree. For an unreachable vertex, however, the value of `pred` is meaningless. In the example `pred[5] = 0`, but there is no path from 0 to 5; this value is left over from the initial assignment in A1. If `print_path(pred, 0, 5)` is called, a path that does not really exist, such as "0 -> 5", is printed. So the caller should first check the condition `dist[t] >= INF` and, if it holds, print "no path."

The trace of the call `print_path(pred, 0, 1)` is in Table 18.6. In the call-stack column, the leftmost frame is the oldest call and the rightmost the newest.

Step	Event	Call stack	Output
1	`t` = 1 ≠ 0; call for `pred[1]` = 3	pp(1)	

Step	Event	Call stack	Output
2	`t` = 3 ≠ 0; call for `pred[3]` = 2	pp(1) pp(3)	
3	`t` = 2 ≠ 0; call for `pred[2]` = 0	pp(1) pp(3) pp(2)	
4	`t` = `s` = 0: the source is printed, return	pp(1) pp(3) pp(2) pp(0)	`0`
5	Back in `print_path(2)` : " -> 2"	pp(1) pp(3) pp(2)	`0 -> 2`
6	Back in `print_path(3)` : " -> 3"	pp(1) pp(3)	`0 -> 2 -> 3`
7	Back in `print_path(1)` : " -> 1"	pp(1)	`0 -> 2 -> 3 -> 1`

Table 18.6: Trace of `print_path(pred, 0, 1)` (`pred = {0, 3, 0, 2, 0, 0}`).

The trace shows that the path is found by walking backward from the target, and that when printing, the call stack puts the order right. The cost of the path is 10 + 15 + 20 = 45, consistent with `dist[1]`. Figure 18.6 shows the predecessor links on the graph.

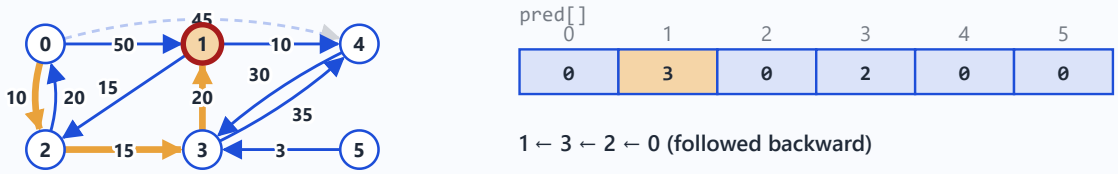


Figure 18.6: The path to 1 from the predecessor array: 1 ← 3 ← 2 ← 0. The highlighted edges are the predecessor links; the chain is followed backward, while the path is printed from source to target.

The cost of the routine is proportional to the length of the path. If the path has *k* edges, *k + 1* calls are made and each does constant work: *O(k)*. The value of *k* is at most *n − 1*. The depth of the call stack is also *k + 1*, so the extra memory is *O(k)* as well.

18.4.3 Printing the path with a stack

The same job can be done without recursion, using an explicit stack. First one walks from the target back to the source and pushes the vertices passed onto the stack. Then the source is printed, and the vertices on the stack are printed in the order they are popped. The stack reverses the order of the predecessor chain.

```c
void print_path_stack(int pred[], int s, int t)
{
    int stack[MAX_VERTICES], top = -1;
    while (t != s) {                /* walk from target to source */
        stack[++top] = t;
        t = pred[t];
    }
    printf("%d", s);
    while (top >= 0)                /* print in reverse order */
        printf(" -> %d", stack[top--]);
}
```

Here the stack consists of an array and an index `top`. The expression `stack[++top] = t` first increments `top` and then writes to that cell; this is a push. `stack[top--]` first reads the cell and then decrements `top`; this is a pop. Since a path has at most $n - 1$ edges, an array of size `MAX_VERTICES` suffices. The program confirmed that the two routines produce the same output for the four reachable targets in the example graph. The recursive version is shorter, while the stack version does not depend on the depth of the call stack. On large graphs, where very long paths are possible, the latter is safer.

18.4.4 Small routines and software quality

The code in this section is deliberately divided into small routines: `min_vertex`, `shortest_path`, `print_path`. Each does a single job and can be tested separately. The value of this division is best understood from flawed examples. On an exam, the code of students who tried to solve the shortest-path question with a single large recursive routine was often neither readable nor working. In some solutions a routine was called as `FindWeight(…);`. The routine returned an `int`, but the returned value was assigned to no variable; the value was computed and lost. The compiler does not count such a line as an error. Only someone reading the code sees the mistake.

Software engineering asks that a solution be not only correct but also modular, simple, readable, reusable, and easy to maintain. Textbooks list these qualities under their standard names. Table 18.7 gives the most common ones with a short description.

Attribute	Meaning	Attribute	Meaning
correctness	the program does what its specification says	reliability	it keeps working correctly under the expected conditions and handles errors
usability	it is easy to learn and use	efficiency	it uses time and memory economically
maintainability	faults are easy to find and fix	modifiability	it is easy to change when requirements change
portability	it runs on other platforms with little change	interoperability	it works together with other programs
integrity	data stays protected against unauthorized access and corruption	flexibility	it adapts to uses beyond the original purpose

Attribute	Meaning	Attribute	Meaning
testability	it is easy to test	traceability	requirements can be traced to the code
understandability	it is easy to read and follow	modularity	it is split into independent parts
expandability	new features are easy to add		

Table 18.7: Some attributes of software quality.

Most of the attributes in the table relate directly to small routines. Because `min_vertex` is a separate routine, switching to the heap version requires changing only that one: modifiability. Because `print_path` is separate, it can be tested on its own with a small predecessor array: testability. Using the return values of functions, `NULL` checks, and comments are also part of reliability and understandability.

18.5 Negative edges and all pairs

18.5.1 Why Dijkstra fails on a negative edge

In real problems an edge weight is not always a length. An edge may represent a gain, an energy recovery, or a discount; in that case the weight is negative. The correctness proof of Dijkstra's algorithm relied on non-negative weights. The four-vertex graph in Figure 18.7 shows what happens when this assumption is broken. The edges are 0 → 1 (4), 0 → 2 (5), 1 → 3 (2), and 2 → 1 (−3).

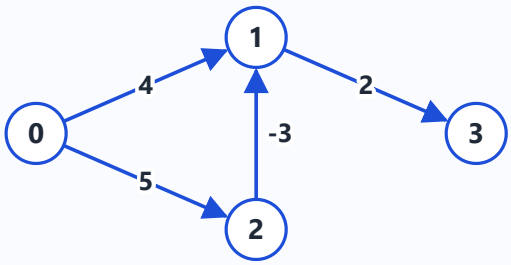


Figure 18.7: A small graph with a negative edge. The weight of the edge 2 → 1 is −3; the path 0 → 2 → 1 (5 − 3 = 2) is shorter than the direct edge 0 → 1 (4).

Dijkstra's algorithm was traced by running a program. At the start the distances are 4 for 1, 5 for 2, and ∞ for 3. In the first round 1, with the smallest value (4), is selected and finalized; through 1 the value of 3 becomes 6. In the second round 2 is selected (5). The edge 2 → 1 opens a path of value 5 − 3 = 2, but 1 is already marked, so no relaxation takes place. The result is in Table 18.8: the values for 1 and 3 are wrong.

Vertex	Dijkstra	Correct	Correct path
0	0	0	—
1	4	2	0 → 2 → 1
2	5	5	0 → 2
3	6	4	0 → 2 → 1 → 3

Table 18.8: The result of Dijkstra's algorithm on the graph with a negative edge, and the correct distances.

The source of the error is the assumption behind the greedy choice. Dijkstra assumes that a vertex, once selected, can never be reached later by a cheaper path. With non-negative weights this is true: the vertices not yet selected are farther away, and a path through them can only get longer. A negative edge, however, can return from a distant vertex and shorten the path. The wrong value of 1 carries over to 3 as well, because the path of 3 goes through 1. Adding the same constant to all weights to get rid of negativity is not a solution either. This transformation penalizes paths with many edges more than paths with one, and changes the shortest path itself.

With negative edges there is one more danger: a **negative cycle**. If a cycle whose weights sum to a negative number is reachable from the source, each trip around the cycle shortens the path a little more. In that case the shortest path is undefined; there is no "smallest" length, and the value decreases without bound. A good algorithm must be able to find and report such a cycle.

18.5.2 The Bellman-Ford algorithm

The algorithm, named after Richard Bellman and Lester Ford, is not greedy; it finalizes no vertex early. Instead it relaxes all the edges over and over. The algorithm keeps the edges in an **edge list**. Each element carries the two endpoints of the edge and its weight:

```c
typedef struct {
    int u, v, w;        /* edge <u,v>, weight w */
} edge;
int BellmanFord(edge edges[], int n, int e, int s, int d[])
{
    int i, j, t, u, v;
    for (i = 0; i < n; i++) d[i] = INF;
    d[s] = 0;
    for (t = 1; t <= n - 1; t++)          /* n-1 rounds */
        for (j = 0; j < e; j++) {
            u = edges[j].u;  v = edges[j].v;
            if (d[u] != INF && d[u] + edges[j].w < d[v])
                d[v] = d[u] + edges[j].w;
        }
    for (j = 0; j < e; j++)               /* round n */
        if (d[edges[j].u] != INF &&
            d[edges[j].u] + edges[j].w < d[edges[j].v]) return -1;
    return 0;              /* -1: negative cycle exists */
}
```

The body of the code is the same as the fragment in the slides; only the routine header and local variables were added around it. `edges` is the edge list, `n` the number of vertices, `e` the number of edges, `s` the source, and `d` the result array.

- **Lines 7–8.** All distances are set to infinity and that of the source to 0. At the start, only the source is known to be reachable.
- **Lines 9–14.** The main loop runs $n - 1$ rounds. In each round all the edges are relaxed in turn. The first part of the condition, `d[u] != INF`, prevents relaxing from a vertex that has not yet been reached. This check is mandatory because of negative edges: `INF + (-3)` is less than infinity, and without the check an unreached vertex would look "reached."
- **Lines 15–17.** Round n: the edges are tried once more, but this time no value is changed. If some edge can still be relaxed, there is a negative cycle and the routine returns −1.
- **Line 18.** If there is no cycle, it returns 0; the array `d` holds the shortest distances.

Why exactly $n - 1$ rounds? If there is no negative cycle, every shortest path can be chosen to be simple, and a simple path has at most $n - 1$ edges. What relaxation does is advance a path edge by edge.

Theorem 18.3 (Number of rounds in Bellman-Ford)

If there is no negative cycle reachable from the source, then after t rounds the value `d` is correct for every vertex that has a shortest path of at most t edges. Therefore $n - 1$ rounds suffice for all vertices.

Proof. Induction on t. For $t = 0$ the only vertex is the source, and $d[s] = 0$ is correct. Let the shortest path to v have t edges and let its last edge be $\langle u, v \rangle$. The part of the path up to u is a shortest path of $t - 1$ edges. By the induction hypothesis, $d[u]$ is correct after $t - 1$ rounds. When the edge $\langle u, v \rangle$ is relaxed in round t, $d[v] \le d[u] + w$, which is the shortest length. Since $d[v]$ is always the length of an actual path, it cannot be smaller than this. ∎

The converse of the theorem explains the cycle test. If there is no negative cycle, after $n - 1$ rounds all values are final and no edge can be relaxed. If an edge is still relaxed in round n, the assumption is broken: a reachable negative cycle exists. The program tried this case on a graph in which the cycle 1 → 2 → 1 sums to −1; the routine returned −1.

The trace on the example graph is in Table 18.9. The order of the edge list is 0 → 1, 0 → 2, 1 → 3, 2 → 1; the result does not depend on this order, but the intermediate values do.

Round	Edge	Computation	`d[0..3]`
—	—	start	0 ∞ ∞ ∞
1	0 → 1 (4)	0 + 4 = 4 < ∞	0 4 ∞ ∞
1	0 → 2 (5)	0 + 5 = 5 < ∞	0 4 5 ∞
1	1 → 3 (2)	4 + 2 = 6 < ∞	0 4 5 6
1	2 → 1 (−3)	5 + (−3) = 2 < 4	0 **2** 5 6
2	0 → 1, 0 → 2	4 ≥ 2, 5 ≥ 5: no change	0 2 5 6
2	1 → 3 (2)	2 + 2 = 4 < 6	0 2 5 **4**
2	2 → 1 (−3)	5 + (−3) = 2 ≥ 2: no change	0 2 5 4
3	all	no edge can be relaxed	0 2 5 4
n.	all	no relaxation → `return 0`	0 2 5 4

Table 18.9: Bellman-Ford trace (source 0, $n = 4$, $e = 4$).

The relaxation in the second round is important. In the first round the value of 3 had become 6 through the old value of 1 (4). When the value of 1 dropped to 2 in the same round, the correct value of 3 could be seen only in the next round. The order of the edges causes this: the edge 1 → 3 comes before 2 → 1 in the list. With a different order the correct values could have been found earlier. The algorithm's guarantee is that $n - 1$ rounds suffice, whatever the order. Figure 18.8 shows the final state of the trace.

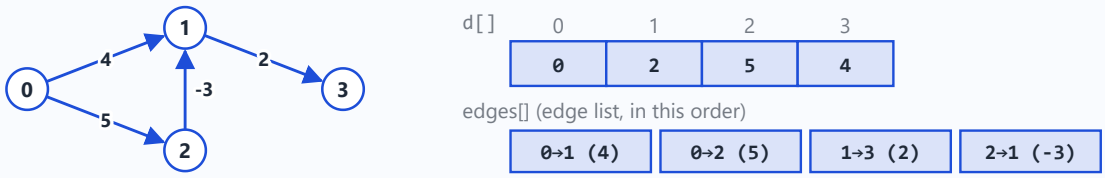


Figure 18.8: When Bellman-Ford finishes: `d = [0, 2, 5, 4]`. On the right the edge list is written in relaxation order.

The cost is clear: in each of the $n - 1$ rounds e edges are tried, and the last round also tries e edges. The total is $O(n \cdot e)$. The extra memory is the array `d`: $O(n)$. In a dense graph $e \approx n^2$, so the time rises to $O(n^3)$. For this reason Bellman-Ford is slower than Dijkstra; it is preferred only when negative weights are present or when a negative cycle is being looked for. In practice, exiting the loop early when no value changes in a round is also a commonly used improvement. In the example the third round changed nothing; with this improvement the algorithm could have stopped there.

18.5.3 All pairs: the Floyd-Warshall algorithm

Sometimes a single source is not enough; as with the distance table on a road map, the shortest length is wanted for every pair of vertices. This is the **all-pairs shortest paths** problem. Running Dijkstra once from every vertex is one solution, but it does not work if there is a negative edge. The algorithm named after Robert Floyd and Stephen Warshall solves the problem with very short code.

The idea of the algorithm is to release, one by one, the intermediate vertices that paths may use.

> **Definition 18.4**
>
> $A^k[i][j]$ is the length of the shortest path from i to j all of whose intermediate vertices lie among $0, 1, \ldots, k$. A^{-1} is the case in which no intermediate vertex may be used; it is the cost matrix itself.

The link between A^k and A^{k-1} comes from two cases. If the shortest path from i to j does not pass through k, its intermediate vertices go only up to $k - 1$ and its length is $A^{k-1}[i][j]$. If it passes through k, the path splits into two parts: from i to k and from k to j. Since there is no negative cycle, neither part uses k as an intermediate vertex, and their lengths are found in A^{k-1}. This gives the relation

$$A^k[i][j] = \min \{ A^{k-1}[i][j], A^{k-1}[i][k] + A^{k-1}[k][j] \}$$

Applying this relation in turn for $k = 0, 1, \ldots, n - 1$ is a **dynamic programming** solution: the big problem is built from smaller subproblems whose results are stored in a table. A^{n-1} is the case in which all vertices may be intermediate, and it is the result itself.

```
void Floyd(int cost[][MAX_VERTICES], int A[][MAX_VERTICES], int n)
{
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            A[i][j] = cost[i][j];
    for (k = 0; k < n; k++)              /* intermediate vertex k */
        for (i = 0; i < n; i++)
            for (j = 0; j < n; j++)
                if (A[i][k] + A[k][j] < A[i][j])
                    A[i][j] = A[i][k] + A[k][j];
}
```

- **Lines 4–6.** The cost matrix is copied into the matrix `A`; this is A^{-1}.
- **Line 7.** The outer loop chooses the intermediate vertex. The order of the loops matters: `k` must be outermost. Each round of `k` uses the whole matrix of the previous round.
- **Lines 8–11.** For every pair `(i, j)`, the path through `k` is tried; if it is shorter, it is written.

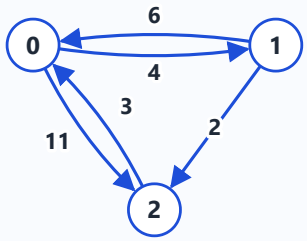
The relation is written with two separate matrices, A^{k-1} and A^{k}, but the code updates a single matrix `A` in place. This is safe. In round k, `A[i][k]` and `A[k][j]` do not change, because in the expression `A[i][k] + A[k][k]` we have `A[k][k] = 0`, and the sum cannot be smaller than `A[i][k]`. The values read during the round are therefore the same as those in A^{k-1}.

The algorithm is traced on Horowitz's three-vertex example. In the cost matrix, 0 → 1 = 4, 0 → 2 = 11, 1 → 0 = 6, 1 → 2 = 2, 2 → 0 = 3, and there is no direct edge from 2 to 1. The program printed the cells that changed in each round of `k`; in every round only one cell changed (Table 18.10).

Matrix	Changed cell	Row 0	Row 1	Row 2
A^{-1}	—	0 4 11	6 0 2	3 ∞ 0
A^{0}	`A[2][1]`: 3 + 4 = 7 < ∞	0 4 11	6 0 2	3 **7** 0
A^{1}	`A[0][2]`: 4 + 2 = 6 < 11	0 4 **6**	6 0 2	3 7 0
A^{2}	`A[1][0]`: 2 + 3 = 5 < 6	0 4 6	**5** 0 2	3 7 0

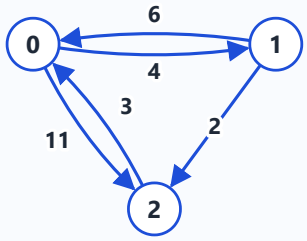
Table 18.10: Floyd-Warshall trace (three vertices). Each row gives the matrix at the end of one round of `k`.

Every change is the discovery of a path. At *k = 0* a path from 2 to 1 through 0 is found: 2 → 0 → 1. At *k = 1*, going from 0 to 2 through 1 is cheaper than the direct edge. At *k = 2*, the path from 1 to 0 through 2 (2 + 3 = 5) is shorter than the direct edge (6). The reason this change can happen only in round *k = 2* is simple: the path uses 2 as an intermediate vertex, and the use of 2 becomes available in this round. Figure 18.9 shows the initial and final matrices.



A (k = −1)

	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0



A (k = 2), result

	0	1	2
0	0	4	6
1	5	0	2
2	3	7	0

Figure 18.9: The beginning and the end of Floyd-Warshall. Top, A^{-1}, that is, the cost matrix; bottom, the all-pairs shortest lengths.

The analysis is short: each of the three nested loops runs n rounds. The time is $\Theta(n^3)$ and does not depend on the number of edges of the graph. The memory is an $n \times n$ matrix. The same routine was also applied to the six-vertex main example. Row 0 of the matrix came out as `0 45 10 25 45 1000`, the same as Dijkstra's result. Row 5 is `58 23 38 3 33 0`: since the only edge leaving 5 goes to 3, all paths from 5 pass through 3. The 1000s in column 5 show that no vertex can reach 5.

Floyd-Warshall allows negative edges but not negative cycles. If there is a negative cycle, a negative value appears on the diagonal at the end of the run: a path of negative length from a vertex to itself has been found. This is a simple way to test for the existence of a cycle.

Common mistake

Adding `INF` like an ordinary number in a graph with negative edges. In the code, the sum `A[i][k] + A[k][j]` is computed without a check. Let `A[i][k]` be infinite (1000) and `A[k][j]` be −5. The sum is 995, which is less than 1000; a path that does not exist is "found." The program tested this on a three-vertex graph: with no path at all from 0 to 2, `A[0][2] = 995` came out. If there are negative edges, the condition must be written as `A[i][k] != INF && A[k][j] != INF && …`. The Bellman-Ford code already makes this check.

18.5.4 Which algorithm, and when?

There is no single "best" algorithm for shortest paths. The choice depends on the form of the problem and on the structure of the graph. Table 18.11 gives a summary for deciding.

Problem	Algorithm	Time	Condition
Unweighted, single source	breadth-first search	$O(n + e)$	all edges equal
Single source, $w \geq 0$	Dijkstra (matrix)	$O(n^2)$	dense graph
Single source, $w \geq 0$	Dijkstra (heap + list)	$O((n + e) \log n)$	sparse graph
Single source, negative edges	Bellman-Ford	$O(n \cdot e)$	detects negative cycles
Directed acyclic graph, single source	relaxation in topological order	$O(n + e)$	no cycle
All pairs	Floyd-Warshall	$O(n^3)$	no negative cycle
All pairs, sparse	n runs of Dijkstra	$O(n\,(n + e) \log n)$	$w \geq 0$

Table 18.11: Types of shortest-path problems and suitable algorithms.

The first row is a reminder: if all edge weights are equal, the shortest path is the one with the fewest edges. Because breadth-first search visits vertices from the source in order of the number of edges, it finds this path in linear time. The fifth row is the subject of the second half of the chapter. If the graph has no cycle, the vertices can be arranged so that the start of every edge comes before its end. If the edges of each vertex are relaxed once in this order, then by the time a vertex is reached, all its predecessors are already final. A single pass suffices even if edge weights are negative. The time computations in activity networks rest on exactly this idea.

18.6 Activity networks and topological order

18.6.1 Projects and activities

A big job is the sum of smaller jobs. The construction of a building starts with laying the foundation; a certain time after the foundation sets, the first floor is poured, and then the second. The electrical wiring cannot be run before the walls are built, but painting and landscaping can go on at the same time. A university program is like this too: graduation comes from passing each of the courses, and some courses can be taken only after other courses. These smaller jobs are called **activities**, and the whole set of activities is called a **project**. A project ends when all its activities have ended.

The project manager's questions are clear. In what order can the jobs be done? When can the project finish at the earliest? Which jobs are critical, that is, which ones delay the whole project by a day if they slip by a day? Which jobs have slack, that is, can slip a little without consequence? To answer these questions, the activities and the before-and-after relations among them are shown in a graph. This graph is called an **activity network**.

Table 18.12 gives fifteen courses of a computer science program and their prerequisites. The example is taken from the book by Horowitz and Sahni; the course names are left in English, as they are there.

Code	Course	Prerequisites	Code	Course	Prerequisites
C1	Programming I	—	C9	Operating Systems	C7, C8
C2	Discrete Mathematics	—	C10	Programming Languages	C7

Code	Course	Prerequisites	Code	Course	Prerequisites
C3	Data Structures	C1, C2	C11	Compiler Design	C10
C4	Calculus I	—	C12	Artificial Intelligence	C7
C5	Calculus II	C4	C13	Computational Theory	C7
C6	Linear Algebra	C5	C14	Parallel Algorithms	C13
C7	Analysis of Algorithms	C3, C6	C15	Numerical Analysis	—
C8	Assembly Language	C3			

Table 18.12: Courses of a computer science program and their prerequisites.

18.6.2 Activity-on-vertex networks and the precedence relation

The natural way to turn the information in the table into a graph is to make every course a vertex. If course *i* is a prerequisite of *j*, a directed edge is drawn from *i* to *j*. The graph with fifteen vertices and fourteen edges is in Figure 18.10.

Definition 18.5

A graph whose vertices are activities and whose directed edges show the **precedence relation** between activities is called an **activity-on-vertex network** (AOV network for short). If activity *i* must finish before *j* can start, the network contains the edge *<i, j>*.

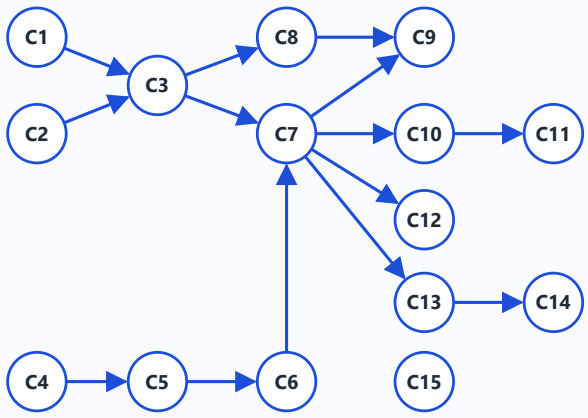


Figure 18.10: The activity-on-vertex network of the courses. Each arrow shows a prerequisite: C1 and C2 are prerequisites of C3; C3 and C6 are prerequisites of C7. C15 has neither a prerequisite nor a successor.

A few terms are needed to name the relations in the network. They resemble family relations, but the direction is determined by the edges.

Definition 18.6

In an AOV network, if there is a directed path from i to j, then i is a **predecessor** of j and j is a **successor** of i. If the path consists of a single edge, that is, if $<i, j>$ is an edge, then i is an **immediate predecessor** of j and j is an **immediate successor** of i.

In the course network, C3 and C6 are the immediate predecessors of C7. C9, C10, C12, and C13 are immediate successors of C7. C14 is a successor of C3 but not an immediate successor: C3 reaches C14 along the path C3 → C7 → C13 → C14. Figure 18.11 highlights the immediate predecessors and successors of C7. The precedence relation has two basic properties.

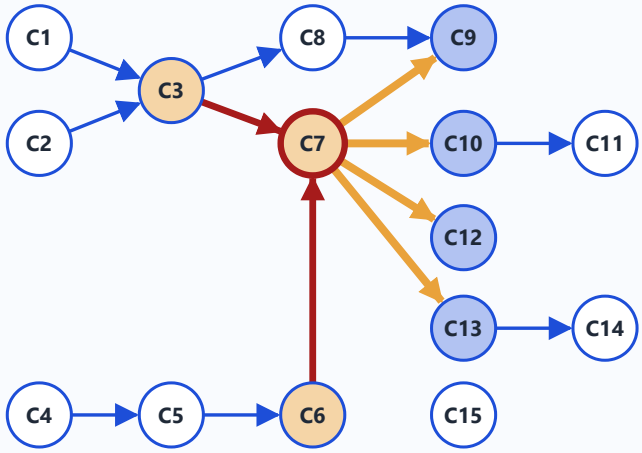


Figure 18.11: C7 and its surroundings. The edges coming from the immediate predecessors (C3, C6) are drawn in thick red, and those going to the immediate successors (C9, C10, C12, C13) in orange.

- **Transitivity.** If i is a predecessor of j and j is a predecessor of k, then i is a predecessor of k. If C1 is a prerequisite of C3 and C3 is a prerequisite of C7, then C1 is also, indirectly, a prerequisite of C7.
- **Acyclicity.** No activity can be its own predecessor. If A is a predecessor of B, then B cannot be a predecessor of A: B cannot start before A finishes, A cannot start before B finishes, and the project never ends. This means the graph contains no directed cycle.

The second property is not merely a theoretical condition; in real projects it is a common planning error. A directed graph that contains no directed cycle is called a **directed acyclic graph** (DAG). A valid AOV network is always a DAG. So when a network is built, it must be checked for cycles.

It is no accident that the course example is woven from courses such as numerical analysis and linear algebra. The roots of topics such as project planning, resource allocation, and shortest paths reach back to operations research. Dropping operations research and numerical analysis courses from curricula leaves students weak in the mathematics of such problems.

18.6.3 Topological order

The course network puts a practical question to a student: in what order can the courses be taken? The order must be one in which the prerequisites of every course come before it.

> **Definition 18.7**
>
> A linear ordering of the vertices of an AOV network in which, whenever *i* is a predecessor of *j*, *i* comes before *j* in the order, is called a **topological order**.

The definition has three consequences. First, a topological order is often not unique. Two activities with no precedence relation between them can be written in either order. Second, if the graph has a directed cycle, there is no topological order: whichever vertex of the cycle is written first, some edge of the cycle points backward. Third, if there is no cycle, at least one topological order exists; the algorithm below is the proof.

Two valid orders for the course network are:

- C1, C2, C4, C5, C3, C6, C8, C7, C10, C13, C12, C14, C15, C11, C9
- C4, C5, C2, C1, C6, C3, C8, C15, C7, C9, C10, C11, C12, C13, C14

It is easy to check whether an order is topological. Find the position of every vertex in the order. Then, for each edge *<i, j>*, check that the position of *i* is smaller than that of *j*. The course network has 14 edges, and both orders satisfy all 14; this check was done with a program. When the `topological_order` routine, written below, was applied to the same network, it produced a third valid order: C15, C4, C5, C6, C2, C1, C3, C8, C7, C13, C14, C12, C10, C11, C9. All three orders are valid; which one comes out depends only on how the algorithm chooses vertices.

Topological order has wide uses. Course planning is only one of them. In a software project, the order in which source files are compiled (the job of the `make` tool), the scheduling of tasks in an operating system, and the order in which cells of a spreadsheet are recalculated are also topological orders. In a spreadsheet, if a cell's formula refers to other cells, those cells must be computed first; if two cells refer to each other, the program reports a "circular reference" error. This is another face of the cycle in an AOV network. The earliest-time computation in activity networks is also done in topological order.

18.6.4 Deleting a vertex with no predecessor

The way to find a topological order rests on a simple observation: the first vertex of the order cannot have any predecessor. So a vertex with no predecessor is chosen and written. Then this vertex and the edges leaving it are deleted from the graph. The remaining graph is also an AOV network, and the same step is repeated on it. When all vertices have been written, the order is complete.

The method is traced on a network of six vertices (Figure 18.12). The edges are *<0,1>, <0,2>, <0,3>, <1,4>, <2,4>, <2,5>, <3,4>, <3,5>*.

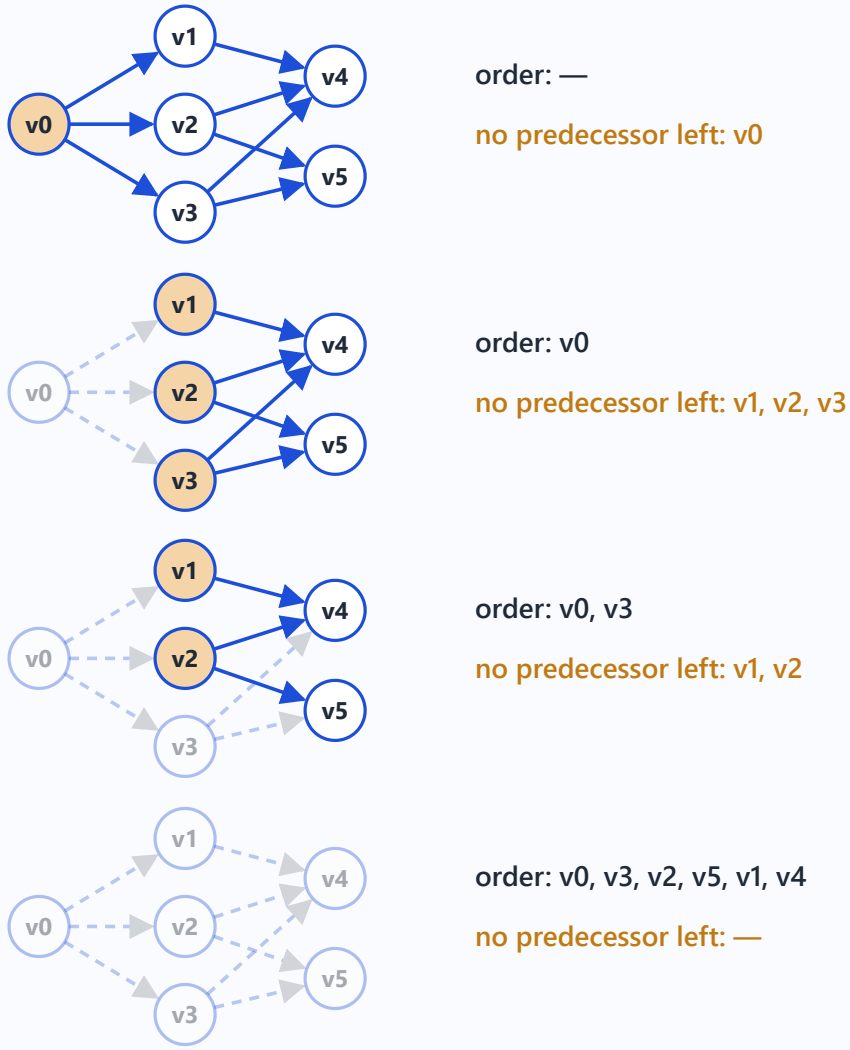


Figure 18.12: The method of deleting a vertex with no predecessor, from top to bottom: (a) only v0 has no predecessor; (b) after v0 is written and its edges deleted, v1, v2, v3 have none left; (c) v3 has been chosen and its edges deleted; (g) all vertices have been written. The orange vertices are the predecessor-free vertices that could be chosen at that moment; the lines on the right give the order so far and the vertices with no predecessor left.

At the start only v0 has no predecessor, and it is written first. When the three edges of v0 are deleted, v1, v2, and v3 are left with no predecessor. Any of the three may be chosen; in the example v3 is chosen. The edges *<3,4>* and *<3,5>* of v3 are deleted. Then v2, then v5, v1, and finally v4 are written. The result is v0, v3, v2, v5, v1, v4. Counting by program showed that this small network has 14 different topological orders; all of them can be reached by making a different choice at each step.

The method also finds cycles. If at some step there are vertices not yet written but none of them is free of predecessors, then in the remaining graph every vertex has at least one predecessor. Walking backward along predecessors, since there are finitely many vertices, some vertex is reached a second

time: this is a directed cycle. The converse is also true: if there is no cycle, at least one predecessor-free vertex is found at every step, and the method writes all vertices. So the method both finds the order and tests whether the network is valid.

For the method to become efficient code, two operations must be fast: finding a predecessor-free vertex and deleting the outgoing edges of a vertex. If at every step all vertices are scanned to find a predecessor-free one, the cost rises to $O(n^2)$. A better way is to keep, in a counter, how many predecessors each vertex still has, and to collect the vertices whose counter drops to zero in a separate place.

18.6.5 Data structure: counted headers

The graph is stored as adjacency lists. Each vertex has a **header** element. Besides the link to the first element of the list, the header has a `count` field. This field holds how many immediate predecessors the vertex has, that is, its **in-degree**.

```
typedef struct node *node_ptr;
typedef struct node {
    int vertex;
    node_ptr link;
} node;
typedef struct {
    int count;      /* for <i,j>: number of edges whose head is j */
    node_ptr link;
} head_node;
head_node head[MAX_VERTICES];
```

- **Line 1.** `node_ptr` is a pointer type that points to a list element. The type name can be declared before the structure is defined; C allows this through an incomplete type. This lets the structure contain a field that points to its own type.
- **Lines 2–5.** The list element: the vertex the edge leads to (`vertex`) and the next element in the list (`link`).
- **Lines 6–9.** The header element: the predecessor counter and the head of the list. The headers sit in an array; the list elements are taken from heap memory with `malloc` .
- **Line 10.** The headers of all vertices. Because it is a global array, all fields are zero when the program starts: the counters are 0 and the links are `NULL` .

When the edges are read, for each *<i, j>*, *j* is added to the list of *i* and `head[j].count` is incremented. The textbook does not write this step out separately; the routine below completes it. It appends the element to the end of the list, so the lists keep the order in which the edges were given.

```
int add_edge(head_node head[], int i, int j)
{
    node_ptr *tail = &head[i].link;   /* address of the link field */
    node_ptr new_node = malloc(sizeof(node));
    if (new_node == NULL)
        return 0;                     /* out of memory */
    new_node->vertex = j;
    new_node->link = NULL;
    while (*tail != NULL)              /* find the end of the list */
        tail = &(*tail)->link;
    *tail = new_node;
    head[j].count++;                 /* j has one more predecessor */
    return 1;
}
```

The interesting part of the routine is the variable `tail`. `tail` points not to an element but to a *link field*; its type is pointer to pointer (`node_ptr *`). At first it holds the address of the field `head[i].link`. If the list is empty, this field is `NULL` and the new element is written directly there. If the list is not empty, at each step the loop moves to the `link` field of the next element and stops at the last link, which is `NULL`. The assignment `*tail = new_node` handles the empty list and the non-empty list with the same line. Without a pointer to a pointer, a separate `if` would be needed for the empty list.

The return value of `malloc` must always be checked. If memory runs out, `malloc` returns `NULL`, and the assignment `new_node->vertex` would cause undefined behavior. The routine reports this to the caller by returning 0. Because insertion is at the end of the list, the cost is *O(d)* if the out-degree of a vertex is *d*. Insertion at the head would take *O(1)*, but the lists would be in the reverse of the edge order, which would change the output of the topological order.

When the edges of the six-vertex network are added in the order *<0,1>, <0,2>, <0,3>, <1,4>, <2,4>, <2,5>, <3,5>, <3,4>*, the structure in Figure 18.13 results. That v5 comes before v4 in the list of v3 is due to the edges being given in this order. The counters are 0, 1, 1, 1, 3, 2: v4 has three predecessors (v1, v2, v3), and v5 has two (v2, v3).

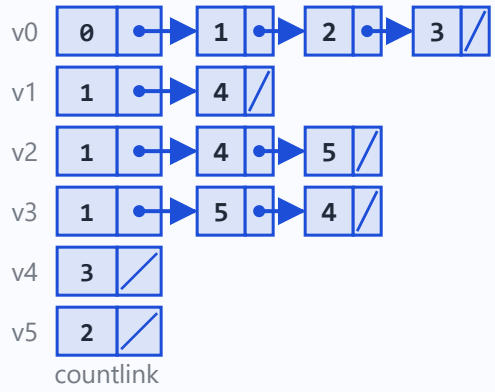


Figure 18.13: The adjacency list with counters for the six-vertex network. On the left the header array (`count` and `link`); on the right the immediate successors of each vertex.

In memory, each list element consists of an `int` and a pointer. On a 64-bit system an `int` is 4 bytes and a pointer is 8. Because a pointer must sit at an address that is a multiple of 8, 4 bytes of padding are added after the `int` , and `sizeof(node)` is 16. The header element is 16 bytes for the same reason. These values were measured on x86-64 with the zig cc compiler. The six-vertex, eight-edge network uses 96 bytes for the headers and 128 bytes for the list elements. The total memory is $O(n + e)$.

When the lists are no longer needed, the memory must be given back. The routine below walks each list from start to end and frees the elements one by one:

```
void free_lists(head_node head[], int n)
{
    int i;
    node_ptr p, next;
    for (i = 0; i < n; i++) {
        for (p = head[i].link; p; p = next) {
            next = p->link;         /* save before free */
            free(p);
        }
        head[i].link = NULL;
        head[i].count = 0;
    }
}
```

The variable `next` in the loop is essential. Accessing `p->link` after `free(p)` means reading freed memory, which is undefined behavior. So the address of the next element is saved before `free` . To check for leaks, `malloc` and `free` were replaced in the tests by small wrappers that count calls. In the chapter's topological-order tests, 24 `malloc` and 24 `free` calls were counted; there is no memory leak. This counting method was used because on Windows the zig compiler could not link the address sanitizer (`-fsanitize=address`).

18.6.6 The `topological_order` routine

The routine collects the predecessor-free vertices on a stack. No separate array is set aside for the stack; the `count` field of a vertex whose counter has dropped to zero is now free and is used as the link of the stack.

```
void topological_order(head_node head[], int n)
{
    int i, j, k, top = -1;
    node_ptr p;
    for (i = 0; i < n; i++)      /* count 0: push on the stack */
        if (!head[i].count) {
            head[i].count = top;  top = i; }
    for (i = 0; i < n; i++) {
        if (top == -1) { printf("cycle"); break; }
        j = top;  top = head[top].count;
        printf("%d ", j);
        for (p = head[j].link; p; p = p->link) {
            k = p->vertex;
            if (--head[k].count == 0) {
                head[k].count = top;  top = k; }
        }
    }
}
```

- **Line 3.** `top` is the vertex at the top of the stack. −1 means the stack is empty, that is, marks its bottom.
- **Lines 5–7.** The first loop pushes all vertices whose counter is 0 onto the stack. The assignment `head[i].count = top` writes the vertex that lies beneath it in the stack into its `count` field. Then `top = i` brings the new vertex to the top. This is the same as the insertion at the head of a stack implemented with a linked list; `count` is used as the link field.
- **Line 8.** The second loop runs *n* rounds; in each round one vertex is written.
- **Line 9.** If the stack is empty when a vertex is about to be written, no predecessor-free vertex is left. Since not all *n* vertices have been written yet, the remaining vertices form a cycle. The routine prints "cycle" and stops.
- **Lines 10–11.** Popping from the stack: the vertex at the top is taken into `j`, and the vertex beneath it (`head[top].count`) becomes the new top. `j` is printed.
- **Lines 12–16.** The list of `j` is walked. The predecessor count of every immediate successor `k` is decreased by one; deleting `j` means deleting the edges leaving `j`. If the counter drops to 0, `k` has no predecessor left and is pushed onto the stack.

The embedded-stack technique is surprising at first. There is no variable called "stack" in the code; only the comment and the variable `top` give away what it is. This is exactly why comments matter: someone reading the code learns from them where and how the stack is used. The technique is similar to the parent array used for sets. There each element holds the number of its parent, and the

links are embedded in an array. Here too the links of the stack are embedded in the header array. A field can serve two separate purposes because the two purposes do not overlap in time. Until the counter drops to 0, the field holds the predecessor count; after it drops, the predecessor count is never needed again.

The expression `--head[k].count == 0` combines two jobs: it first decrements the counter, then compares the new value with 0. It means the same as the pair `count--; if (!count)` in the textbook. The prefix decrement operator matters here. If the postfix form were used (`head[k].count-- == 0`), the comparison would be made with the old value. When the last predecessor is deleted, the old value is 1, so no successor would ever enter the stack; in the example the output would be cut off with `0 cycle`.

Common mistake

Calling `topological_order` a second time on the same header array. When the routine finishes, the `count` fields hold not predecessor counts but leftovers of the stack links. In the example the counters were left as −1, −1, 1, 2, −1, 1 after the routine. To run it again, the counters must be recomputed from the lists or the lists must be rebuilt.

18.6.7 Code trace

The trace of the call `topological_order(head, 6)` is in Table 18.13. In the stack column the top is on the left. The program printed `0 3 2 5 1 4` on the same network.

Event	`count[]`	Stack	Output
First loop: v0 has counter 0 → push	−1 1 1 1 3 2	0	
Pop: v0	−1 1 1 1 3 2	—	0
<0,1>: v1 dropped to 0 → push	−1 **−1** 1 1 3 2	1	0
<0,2>: v2 dropped to 0 → push	−1 −1 **1** 1 3 2	2 1	0
<0,3>: v3 dropped to 0 → push	−1 −1 1 **2** 3 2	3 2 1	0
Pop: v3	−1 −1 1 2 3 2	2 1	0 3
<3,5>: counter of v5 is 1; <3,4>: counter of v4 is 2	−1 −1 1 2 2 1	2 1	0 3
Pop: v2	−1 −1 1 2 2 1	1	0 3 2
<2,4>: counter of v4 is 1; <2,5>: v5 dropped to 0 → push	−1 −1 1 2 1 **1**	5 1	0 3 2
Pop: v5 (its list is empty)	−1 −1 1 2 1 1	1	0 3 2 5
Pop: v1	−1 −1 1 2 1 1	—	0 3 2 5 1
<1,4>: v4 dropped to 0 → push	−1 −1 1 2 **−1** 1	4	0 3 2 5 1
Pop: v4	−1 −1 1 2 −1 1	—	0 3 2 5 1 4

Table 18.13: Trace of `topological_order(head, 6)`. The counters are in the order v0…v5; the counter of a vertex on the stack points to the vertex beneath it.

Two details of the table deserve attention. When a vertex is pushed, the current value of `top` is written into its counter: when v2 is pushed the top is v1, so `count[2] = 1`; when v3 is pushed the top is v2, so `count[3] = 2`. Also, after v0 is popped the stack is empty, but it is immediately filled with the successors of v0. The cycle test is made only at the *start* of a round; a temporary emptiness inside a round does not count as a cycle.

The output order is the same as the order found by hand in the figure. The choice of v3 in the hand version was arbitrary; in the code it follows from the stack's last-in, first-out rule. Because v1, v2, v3 enter the stack in this order, v3 comes out first. Figure 18.14 shows two moments of the trace.

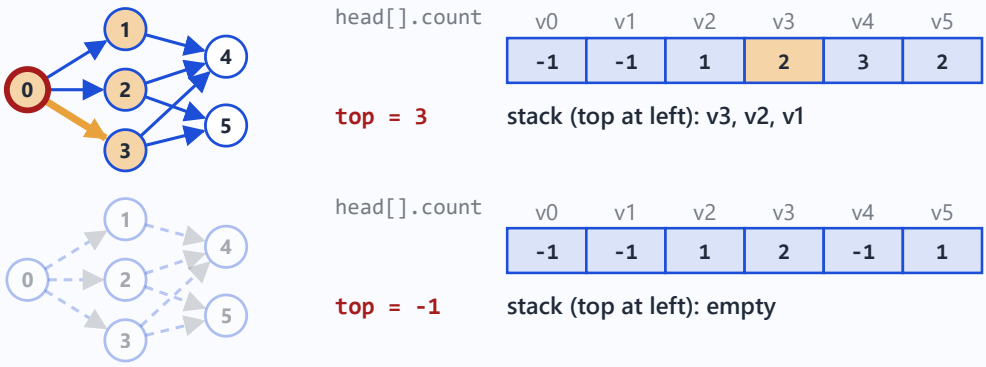


Figure 18.14: Two moments from the trace of `topological_order`. Top: the list of v0 is finished; v1, v2, v3 are on the stack, and the counter fields hold the links of the stack. Bottom: all vertices have been written and all edges deleted.

18.6.8 Cycle detection, analysis, and the queue version

A cycle means two activities are each other's predecessors, and it is a planning error: if there is a cycle, the project never ends. How the routine catches a cycle was tested on a small network. The edges are *<0,1>, <1,2>, <2,1>, <2,3>*; 1 and 2 are each other's predecessors. The routine first writes 0. The counter of 1 drops from 2 to 1 and does not reach zero, because the edge from 2 is still there. At the start of the second round the stack is empty, but only one vertex has been written. The output is `0 cycle`.

The cost of the routine comes from two loops. The first loop looks at the n headers once: $O(n)$. The second loop runs at most n rounds. In each round a vertex is taken from the stack and its list is walked. The list of j has d_j elements, so the inner `for` runs as many times as the out-degree. Since every vertex is taken from the stack at most once, the total number of rounds of the inner loop is the sum of all out-degrees. In a directed graph this sum equals the number of edges. The result is

$$O(n + d_0 + d_1 + \cdots + d_{n-1}) = O(n + e)$$

The number of edges varies with how the graph is connected. In a sparse network e is of the order of n and the time is linear. In a very dense network e can reach the order of n^2. Since every edge must be read at least once, no algorithm can be faster than $O(n + e)$; this routine is optimal.

A stack is not essential for collecting the predecessor-free vertices; any container will do. If a queue is used, the algorithm is named after Kahn (Arthur Kahn, 1962). The version below implements the queue with a simple array:

```
void queue_order(head_node head[], int n)
{
    int queue[MAX_VERTICES], front = 0, rear = 0, i, j, k;
    node_ptr p;
    for (i = 0; i < n; i++)
        if (!head[i].count)
            queue[rear++] = i;       /* enqueue */
    while (front < rear) {
        j = queue[front++];             /* dequeue */
        printf("%d ", j);
        for (p = head[j].link; p; p = p->link) {
            k = p->vertex;
            if (--head[k].count == 0)
                queue[rear++] = k;
        }
    }
    if (rear < n)
        printf("cycle");
}
```

In this version the cycle test is made at the end: if the number of vertices that entered the queue is less than *n*, the counters of some vertices never dropped to zero. On the same six-vertex network the queue version printed `0 1 2 3 5 4`. This time v1, v2, v3 come out in the order they entered. While v3 is processed, the counter of v5 drops to zero first and then that of v4, because v5 is at the front of v3's list. The two outputs differ, but both are valid topological orders. The type of container only chooses among equally valid orders. The queue version also printed `0 cycle` on the network with a cycle, and no memory leak was seen in any test of the two versions.

18.7 Activity-on-edge networks and the critical path

18.7.1 The activity-on-edge network

An AOV network shows the order of jobs but not their durations. To compute the project duration, a second model is used. In this model the activities are placed on the edges, and the moments that determine when activities finish are placed on the vertices.

Definition 18.8

A directed acyclic graph whose directed edges represent activities, whose edge weights represent the durations of the activities, and whose vertices represent **events** is called an **activity-on-edge network** (AOE network for short). An event is the moment at which all activities entering it have finished; the activities leaving the event can start only after this moment.

An event is an intermediate milestone that must be reached for the project to be completed. The job that has to finish is expressed by a vertex: "foundation done," "rough construction done." The network has a single start event (a vertex with no incoming edge) and a single finish event (a vertex with no outgoing edge). The start is the moment the project begins, and the finish is the moment it ends.

For the rest of the chapter the network in Figure 18.15 is used. The network has nine events (v0–v8) and eleven activities (a0–a10). v0 is the start event and v8 is the finish event.

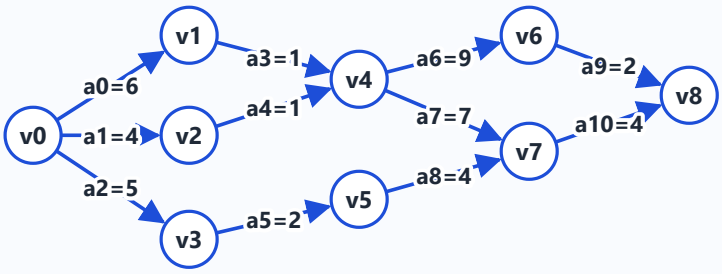


Figure 18.15: An AOE network with nine events and eleven activities. The edge label gives the name and duration (days) of the activity: a0 = 6 is the 6-day activity from v0 to v1.

The network shows parallelism directly. a0, a1, and a2 can start at the same time, on the first day of the project; the only condition of all three is the start event. a6 and a7 leave the event v4. a3 and a4 enter v4. So a6 and a7 cannot start until both a3 and a4 have finished. Similarly, a7 and a8 enter v7; a10 starts only when these two are done. The list of activities is in Table 18.14.

Activity	a0	a1	a2	a3	a4	a5	a6	a7	a8	a9	a10
Edge	0→1	0→2	0→3	1→4	2→4	3→5	4→6	4→7	5→7	6→8	7→8
Time	6	4	5	1	1	2	9	7	4	2	4

Table 18.14: The activities of the AOE network.

18.7.2 From AOV to AOE: the dummy activity

An AOV network can be converted into an AOE network, but the conversion is not always a one-to-one mapping of edges. In the AOV network of Figure 18.16, activity 4 waits for both 2 and 3. In the AOE network, 2 and 3 are edges; for 4 to be able to start, both edges must arrive at the same event. If

the two edges end at different events, these events are joined by an edge of duration 0.

Definition 18.9

An edge of duration 0 added to an AOE network solely to express a precedence relation is called a **dummy activity**.

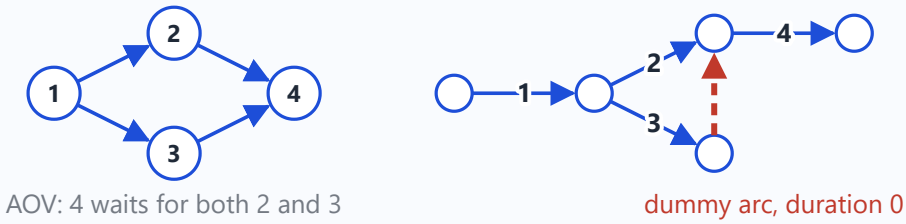


Figure 18.16: On the left the AOV network: 4 waits for both 2 and 3. On the right the same relation in an AOE network: the dashed red edge is the dummy activity of duration 0.

A dummy activity may be needed in the main example too. If a6 and a7 were required to start after a5 finishes, in addition to a3 and a4, an edge would be needed from the event v5, where a5 finishes, to the event v4, where a6 and a7 start. This edge is not a job but only a waiting rule: the dummy activity *<v5, v4>*, duration 0.

Drawing a network with the fewest dummy activities is a problem that looks easy but is hard. It has been shown to be NP-hard; that is, no known algorithm solves it efficiently on large networks. It is a research topic, studied at thesis level in the field concerned with legible drawing of graphs (graph drawing).

AOE networks are the basis of the classical methods of project management. PERT (Program Evaluation and Review Technique) was developed in 1958 for the US Navy's Polaris missile program. CPM (Critical Path Method) arose in industry in the same years, for planning maintenance and construction projects. RAMPS (Resource Allocation and Multi-Project Scheduling) deals with sharing resources among several projects. Most of today's project-management software performs the computations of these methods.

18.7.3 The critical path

Because activities can run in parallel, the project duration is not the sum of all durations. The sum is 6 + 4 + 5 + 1 + 1 + 2 + 9 + 7 + 4 + 2 + 4 = 45 days, but the project finishes in far less time. The finish event occurs when the activities on every path leading to it have finished. So the project duration is the length of the *longest* of the paths from start to finish.

Definition 18.10

In an AOE network, the longest path from the start event to the finish event is called the **critical path**. The length of the critical path is the shortest time in which the project can finish. A network may have more than one critical path.

The example network has two critical paths: v0-v1-v4-v6-v8 (6 + 1 + 9 + 2 = 18) and v0-v1-v4-v7-v8 (6 + 1 + 7 + 4 = 18). The project finishes in 18 days at the earliest. The other paths are shorter; for example the path v0-v3-v5-v7-v8 takes 5 + 2 + 4 + 4 = 15 days. Figure 18.17 shows the critical paths with thick lines.

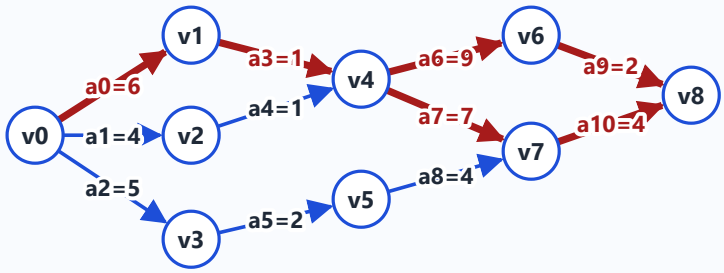


Figure 18.17: The critical paths: v0-v1-v4-v6-v8 and v0-v1-v4-v7-v8, length 18. The two paths share everything up to v4.

That "shortest time" and "longest path" are the same thing looks contradictory at first. There is no contradiction: the project cannot finish sooner than the longest chain of waiting, but when that chain is done, no other job is left waiting. In a general graph, finding the longest simple path is NP-hard. An AOE network, however, is acyclic, and in an acyclic graph the longest path is found with a single pass in topological order. The computations below do exactly this.

The concept of the critical path tells the manager two things. If an activity on the critical path slips by a day, the project slips by a day. The foremen of these jobs are told: "Do not let 4 days stretch to 5; I would be glad if it finished in 3." For activities not on the critical path, a certain amount of delay does not affect the project. To find out what this amount is, the earliest and latest start times of every activity are computed. The problem is an operations research problem.

18.8 Earliest and latest times

These computations look simple on paper. But running them by hand takes care: if one mixes up which value is final at which moment and which value may still change, the result is spoiled. That is why the computations below are given step by step in tables.

18.8.1 The earliest start time

An activity can start as soon as its tail event occurs. The earliest start time of the activity a_i = <k, m> is therefore equal to the earliest time of event *k*. The earliest times of events are kept in the array `earliest`.



Figure 18.18: The activity a_i goes from event *k* to event *m*.

For example, the tail of a1 is v0, and it starts on day 0 at the earliest. The tail of a5 is v3; only a2 enters v3, and it takes 5 days. a5 starts on day 5 at the earliest. The tail of a8 is v5; a5, which starts on day 5 and lasts 2 days, enters v5, so a8 starts on day 5 + 2 = 7 at the earliest.

When several activities enter an event, the situation changes. a3 and a4 enter v4. a3 starts on day 6 and finishes on day 6 + 1 = 7. a4 starts on day 4 and finishes on day 4 + 1 = 5. The event v4 waits for both to finish; so it waits for the one that finishes last, that is, day 7. In general the rule is

> earliest[m] = max { earliest[k] + duration(<k, m>) : k ∈ P(m) }

where *P(m)* is the set of immediate predecessors of *m*. The value of the start event is 0. 0 stands for the day the project begins; if the project starts on May 8, day 0 is May 8. If the start were given 10, all values would shift by 10 days.

The formula has one condition: when *earliest[m]* is computed, the values of the predecessors must be final. This means the computation must be done in **topological order**. So the `topological_order` routine, with a small addition, performs this computation too. A `duration` field holding the duration of the activity is added to the list element:

```
typedef struct node *node_ptr;
typedef struct node {
    int vertex;
    int duration;           /* duration of the activity */
    node_ptr link;
} node;
typedef struct {
    int count;
    node_ptr link;
} head_node;
int earliest[MAX_VERTICES], latest[MAX_VERTICES];
```

With this structure the element size does not change: `vertex` and `duration` take 4 bytes each and fill the place of the earlier padding; `sizeof(node)` is still 16. The edge-insertion routine is also extended to write only the duration:

```
int add_edge(head_node head[], int i, int j, int duration)
{
    node_ptr *tail = &head[i].link;
    node_ptr new_node = malloc(sizeof(node));
    if (new_node == NULL)
        return 0;
    new_node->vertex = j;
    new_node->duration = duration;
    new_node->link = NULL;
    while (*tail != NULL)
        tail = &(*tail)->link;
    *tail = new_node;
    head[j].count++;
    return 1;
}
```

In the inner loop of `topological_order`, before the counter is decremented, these two lines are executed:

```
/* inside topological_order, before the count is decremented: */
if (earliest[k] < earliest[j] + p->duration)
    earliest[k] = earliest[j] + p->duration;
```

`j` is the vertex just taken from the stack and written, and `k` is one of its immediate successors. If the path to `k` through `j` requires a longer wait, the value of `k` is increased. This is exactly the relaxation of Dijkstra's algorithm; the only differences are the direction of the `<` comparison and that the value sought is the largest. `earliest` is initially filled with 0s. Since all durations are non-negative, 0 is a valid lower bound for every event. The complete routine with the additions is:

```
void compute_earliest(head_node head[], int n)
{
    int i, j, k, top = -1;
    node_ptr p;
    for (i = 0; i < n; i++)
        earliest[i] = 0;
    for (i = 0; i < n; i++)
        if (!head[i].count) {
            head[i].count = top;  top = i; }
    for (i = 0; i < n; i++) {
        if (top == -1) { printf("cycle"); break; }
        j = top;  top = head[top].count;   /* pop from the stack */
        printf("%d ", j);
        for (p = head[j].link; p; p = p->link) {
            k = p->vertex;
            if (earliest[k] < earliest[j] + p->duration)
                earliest[k] = earliest[j] + p->duration;
            if (--head[k].count == 0) {
                head[k].count = top;  top = k;  /* push on the stack */
            }
        }
    }
}
```

- **Lines 5–9.** The array is initialized and the predecessor-free vertices are pushed onto the stack. This part is the same as the first loop of `topological_order`.
- **Lines 12–13.** An event is taken from the stack. From this moment on `earliest[j]` never changes again, because all predecessors of `j` were processed earlier.
- **Lines 16–17.** For every activity leaving `j`, the value of the successor is increased if necessary.
- **Lines 18–20.** If the last predecessor of the successor has also been processed, the successor enters the stack. The order of the lines matters: the value update must be done before the counter is decremented. Otherwise, by the time the vertex enters the stack and is written in another round, the contribution from the last edge might not have been added yet.

18.8.2 Tracing the forward pass

Because the computation advances from start to finish, it is called the **forward stage**. The network lists are built in the order of Table 18.14: the list of v0 is v1, v2, v3; the list of v4 is v6, v7. Table 18.15 gives the state after each event is written. The rows were compared with the program's output.

State	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	Stack
Start	0	0	0	0	0	0	0	0	0	0
Output v0	0	6	4	5	0	0	0	0	0	3, 2, 1
Output v3	0	6	4	5	0	7	0	0	0	5, 2, 1
Output v5	0	6	4	5	0	7	0	11	0	2, 1

State	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	Stack
Output v2	0	6	4	5	5	7	0	11	0	1
Output v1	0	6	4	5	7	7	0	11	0	4
Output v4	0	6	4	5	7	7	16	14	0	7, 6
Output v7	0	6	4	5	7	7	16	14	18	6
Output v6	0	6	4	5	7	7	16	14	18	8
Output v8	0	6	4	5	7	7	16	14	18	—

Table 18.15: Computation table for `earliest`. Each row is the state after the named event has been written and its list processed; the top of the stack is on the left.

The most important lesson of the table is which values are permanent. Only the earliest times of events already written to the output are final; from then on they never change. The value of an unwritten event is only a lower bound. When v5 was processed, `earliest[7]` was set to 11. But v7 had one more predecessor: v4. When v4 was processed, 7 + 7 = 14 was found and the value rose to 14. Similarly, the value of v4 became first 5 through v2 and then 7 through v1. The cells that changed are highlighted in the table.

In the last row `earliest[8] = 18`: the project finishes in 18 days at the earliest. This is the length of the critical path. Two activities enter v8: a10 (14 + 4 = 18) and a9 (16 + 2 = 18). Both give 18; this is why there are two critical paths. Figure 18.19 shows two moments of the forward pass.

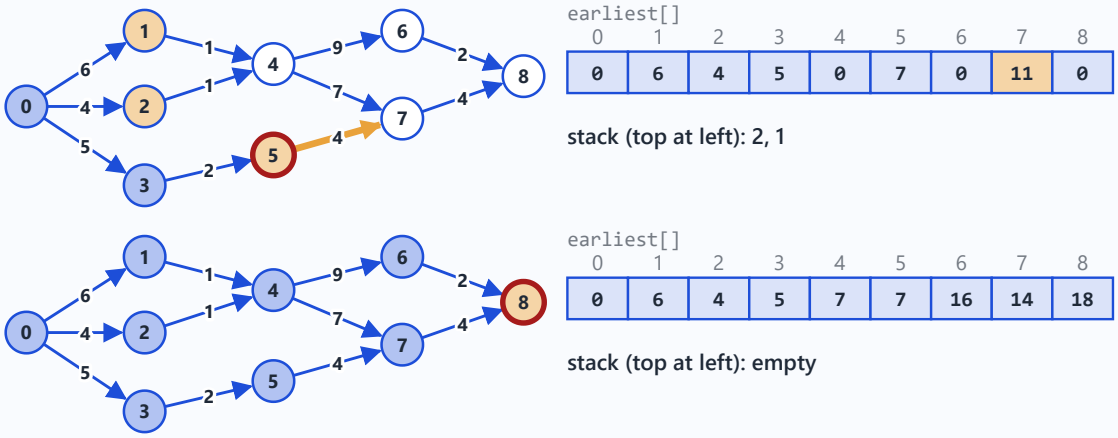


Figure 18.19: Two moments from the forward pass. Top: the list of v5 is being processed: `earliest[7]` becomes 11, but this value is not final. Bottom: all events have been written; the final state of the array is 0 6 4 5 7 7 16 14 18.

18.8.3 The latest start time

The earliest times answer the question "when can we start at the soonest?" The manager's second question is "when must we start at the latest, without delaying the project?" This time the computation starts at the finish and walks backward. The latest time of the finish event is taken equal

to its earliest time: if the project can finish in 18 days, it must finish on day 18. So *latest[8] = earliest[8] = 18*.

The activity a_i = <*k, m*> must start at *latest[m] − duration* at the latest; otherwise event *m* is late. a10 (v7 → v8, 4 days) must start by day 18 − 4 = 14 at the latest. a9 (v6 → v8, 2 days) must start by day 18 − 2 = 16 at the latest. Hence *latest[7] = 14* and *latest[6] = 16*.

If several activities leave an event, the event must meet the activity that has to start earliest. So the smallest value is taken:

> latest[k] = min { latest[m] − duration(<k, m>) : m ∈ S(k) }

where *S(k)* is the set of immediate successors of *k*. v0 is a good example. Three activities leave v0. For a0 we get 6 − 6 = 0, for a1 6 − 4 = 2, for a2 8 − 5 = 3. The smallest is 0, so *latest[0] = 0*. Had the largest, 3, been chosen, a0 would start three days late, v1 would slip to day 9, v4 to day 10, v6 to day 19, and the project would stretch to 21 days. The largest value is taken in the forward pass and the smallest in the backward pass; confusing the two is the most common mistake in such computations.

The computation needs the *successors* of every event, and the values of the successors must become final first. The easiest way is to reverse all the arrows. In the reversed graph, successors turn into predecessors, and the same topological-order routine can be used. The reversed graph is obtained by building the reverse of the adjacency list:

```
int build_reverse(head_node head[], head_node rev[], int n)
{
    int i;
    node_ptr p;
    for (i = 0; i < n; i++) {
        rev[i].count = 0;
        rev[i].link = NULL;
    }
    for (i = 0; i < n; i++)
        for (p = head[i].link; p; p = p->link)      /* <i, p->vertex> */
            if (!add_edge(rev, p->vertex, i, p->duration))
                return 0;
    return 1;
}
```

For every edge <*i, j*>, the routine adds *i* to the reverse list of *j*, with the same duration. `add_edge` also increments the `count` field in the reverse list; this counter is the in-degree in the reversed graph, that is, the out-degree in the original graph. The routine touches every edge once and finishes in *O(n + e)*; the extra cost of appending to the end of a list is also bounded in total by the number of edges. The reverse lists are built only from the `link` and `duration` fields, either before the forward pass spoils the `count` fields or independently of it.

The loop of the backward pass is the same as that of the forward pass; only the relaxation line changes:

```
if (latest[k] > latest[j] - p->duration)
    latest[k] = latest[j] - p->duration;
```

Here `j` is the event written in the reverse list, and `k` is one of its predecessors in the original graph. `latest` is initially filled with the project duration, that is, 18, and values only decrease. The complete routine is:

```
void compute_latest(head_node head[], int n, int finish)
{
    int i, j, k, top = -1;
    node_ptr p;
    for (i = 0; i < n; i++)
        latest[i] = finish;
    for (i = 0; i < n; i++)
        if (!head[i].count) {
            head[i].count = top;  top = i; }
    for (i = 0; i < n; i++) {
        if (top == -1) { printf("cycle"); break; }
        j = top;  top = head[top].count;   /* reverse list */
        printf("%d ", j);
        for (p = head[j].link; p; p = p->link) {
            k = p->vertex;                        /* k: predecessor */
            if (latest[k] > latest[j] - p->duration)
                latest[k] = latest[j] - p->duration;
            if (--head[k].count == 0) {
                head[k].count = top;  top = k;
            }
        }
    }
}
```

18.8.4 Tracing the backward pass

In the reversed graph the only vertex with no predecessor is v8; the computation starts from it. Because the reverse lists are built by scanning the original edges in order of *i*, the reverse list of v8 is v6, v7; that of v7 is v4, v5; that of v4 is v1, v2. Table 18.16 gives the trace; the values were verified with a program.

State	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	Stack
Start	18	18	18	18	18	18	18	18	18	8
Output v8	18	18	18	18	18	18	16	14	18	7, 6
Output v7	18	18	18	18	7	10	16	14	18	5, 6

State	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	Stack
Output v5	18	18	18	8	7	10	16	14	18	3, 6
Output v3	3	18	18	8	7	10	16	14	18	6
Output v6	3	18	18	8	7	10	16	14	18	4
Output v4	3	6	6	8	7	10	16	14	18	2, 1
Output v2	2	6	6	8	7	10	16	14	18	1
Output v1	0	6	6	8	7	10	16	14	18	0
Output v0	0	6	6	8	7	10	16	14	18	—

Table 18.16: Computation table for `latest` (on the reversed graph). Each row is the state after the named event has been written and its reverse list processed.

`latest[0]` changed three times: 8 − 5 = 3 from v3, 6 − 4 = 2 from v2, 6 − 6 = 0 from v1. Each update brings a tighter requirement; the last one says that v0 has no room for delay. While v6 is processed, 16 − 9 = 7 is found for v4; this equals the value found through v7 (14 − 7 = 7), and the array does not change. Both successors of v4 are critical.

The order of the backward pass (8 7 5 3 6 4 2 1 0) is a topological order of the reversed graph. Reading the order of the forward pass (0 3 5 2 1 4 7 6 8) backward does the same job: the reverse of a topological order is a topological order of the reversed graph. A script found the same `latest` values with this second order too. What matters is that when an event is processed, all its successors are final; in an arbitrary order there is no such guarantee.

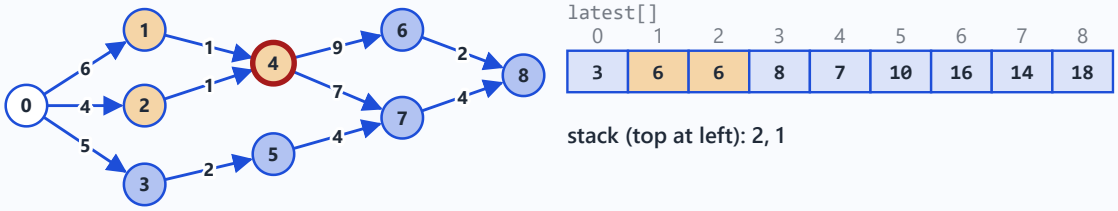


Figure 18.20: In the backward pass after v4 is processed: the latest times of v1 and v2 have become 7 − 1 = 6, and both are on the stack.

18.8.5 Slack and critical activities

Two times are now known for every activity. For a_i = <*k*, *m*>, the earliest start time is *earliest[k]* and the latest start time is *latest[m] − duration*. The difference between them is the number of days the activity can be shifted without delaying the project.

Definition 18.11

The **slack** of the activity $a_i = \langle k, m \rangle$ is the difference between its latest and earliest start times: $latest[m] - duration(a_i) - earliest[k]$. An activity whose slack is 0 is called a **critical activity**.

A critical activity must start at its earliest time; a one-day delay in it delays the whole project by a day. All the activities of the example network are in Table 18.17. The table is the same as the program's output.

Activity	Edge	Time	Earliest	Latest	Slack
a0	(0,1)	6	0	0	**0: critical**
a1	(0,2)	4	0	2	2
a2	(0,3)	5	0	3	3
a3	(1,4)	1	6	6	**0: critical**
a4	(2,4)	1	4	6	2
a5	(3,5)	2	5	8	3
a6	(4,6)	9	7	7	**0: critical**
a7	(4,7)	7	7	7	**0: critical**
a8	(5,7)	4	7	10	3
a9	(6,8)	2	16	16	**0: critical**
a10	(7,8)	4	14	14	**0: critical**

Table 18.17: Earliest and latest start times and slacks of the activities.

An example: the earliest start time of a4 is 4 and its latest start time is 7 − 1 = 6. a4 can be two days late and the project still finishes in 18 days. The critical activities are a0, a3, a6, a7, a9, and a10. They form two paths from v0 to v8, and these paths are the critical paths (Figure 18.21). Project-management software shows the distribution of slack with bar charts; the manager can shift resources from jobs with slack to critical jobs.

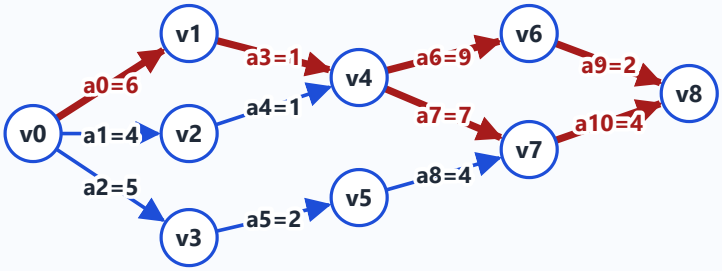


Figure 18.21: The critical activities, whose slack is 0, are drawn with thick red edges; their labels are also highlighted.

The slack in the table is the **total float**. Total float is shared among the activities on the same path. a2, a5, and a8 are on the same path, and all three have slack 3. But this does not mean that all three can be late by three days each. If a2 is three days late, the allowance of a5 and a8 is used up; both become critical. Slack can be spent once along a path.

For this a second measure is used. The **free float** is the slack an activity can use without delaying the earliest time of its successors: *earliest[m] − earliest[k] − duration*. For a2 it is 5 − 0 − 5 = 0, for a5 it is 7 − 5 − 2 = 0, and for a8 it is 14 − 7 − 4 = 3. That is, the slacks of a2 and a5 are not their own; they come from a8 on the same path. a8, on the other hand, can use its three days without affecting anyone. The free float of a4 is 7 − 4 − 1 = 2.

18.8.6 Finding the critical path in a program

Once the two arrays have been computed, printing the slack of every activity takes a single double loop. The lists of the original graph are walked; each element is an activity.

```
void slacks(head_node head[], int n)
{
    int i;
    node_ptr p;
    /* after earliest[] and latest[] have been computed */
    for (i = 0; i < n; i++)
        for (p = head[i].link; p; p = p->link) {
            int early = earliest[i];
            int late   = latest[p->vertex] - p->duration;
            printf("<%d,%d> early=%d late=%d slack=%d%s\n",
                   i, p->vertex, early, late, late - early,
                   late == early ? " CRITICAL" : "");
        }
}
```

The first four lines of the output are:

```
<0,1> early=0 late=0 slack=0 CRITICAL
<0,2> early=0 late=2 slack=2
<0,3> early=0 late=3 slack=3
<1,4> early=6 late=6 slack=0 CRITICAL
```

The remaining lines are identical to Table 18.17. By keeping only the activities with slack 0, the routine also yields the critical path: the critical activities form one or more paths from v0 to v8. The `?:` operator in `printf` selects a string conditionally: if the slack is 0, `" CRITICAL"` is printed; otherwise an empty string.

The cost of the whole computation is the sum of its parts. The forward pass is *O(n + e)*, building the reverse list is *O(n + e)*, the backward pass is *O(n + e)*, and the slack scan is *O(n + e)*. The total is *O(n + e)*; however large the project, the computation finishes in linear time. In the chapter's tests, 22 `malloc` and 22 `free` calls were counted for the forward and reverse lists; there is no leak.

Common mistake

Taking the smallest value for the earliest time and the largest for the latest time. The correct rule is the reverse. An event waits for all the jobs entering it to finish, so the earliest time is the largest value. An event must meet all the jobs leaving it, so the latest time is the smallest value. In code this is the direction of the comparison signs: increasing with `<` in the forward pass, decreasing with `>` in the backward pass.

18.8.7 Application: an exam question

These computations are often asked in exams, and students are expected to carry them out step by step on paper. It is a topic that seems understood but is often gotten wrong when run by hand. The example below is solved with all its steps.

Example 18.1

For the AOE network in Figure 18.22, write the topological order and the stack at every step; fill in the tables for `earliest` and `latest`; find the slack of every activity and identify the critical path. The lists are built in order of activity number.

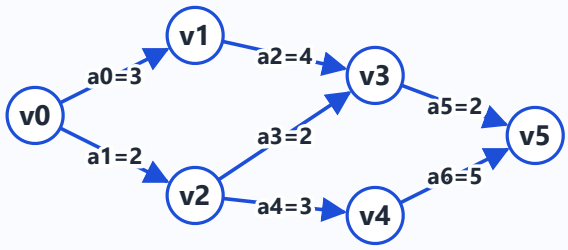


Figure 18.22: The network of the exam question: six events, seven activities. a0 = <0,1> (3), a1 = <0,2> (2), a2 = <1,3> (4), a3 = <2,3> (2), a4 = <2,4> (3), a5 = <3,5> (2), a6 = <4,5> (5).

Forward pass. The counters for v0…v5 are 0, 1, 1, 2, 1, 2. Only v0 is on the stack. When v0 is written, the values of v1 and v2 become 3 and 2; both enter the stack, and the top is v2. When v2 is written, we get 2 + 2 = 4 for v3 and 2 + 3 = 5 for v4. The counter of v3 drops to 1; v4 enters the stack. When v4 is written, we get 5 + 5 = 10 for v5. Then v1 is written and the value of v3 rises to 3 + 4 = 7; v3 enters

the stack. When v3 is written, we get 7 + 2 = 9 for v5, but since 9 < 10 the value does not change. Finally v5 is written. The topological order is 0 2 4 1 3 5, and the result is `earliest = [0, 3, 2, 7, 5, 10]`. The project finishes in 10 days at the earliest.

Backward pass. `latest` starts with 10s. In the reversed graph we start from v5: we get 10 − 2 = 8 for v3 and 10 − 5 = 5 for v4. When v4 is processed, we get 5 − 3 = 2 for v2. When v3 is processed, we get 8 − 4 = 4 for v1; for v2 we get 8 − 2 = 6, but since 2 is smaller, the value stays 2. When v2 is processed, we get 2 − 2 = 0 for v0, and when v1 is processed, 4 − 3 = 1, and 0 is kept. The order is 5 4 3 2 1 0, and the result is `latest = [0, 4, 2, 8, 5, 10]`.

Activity	a0	a1	a2	a3	a4	a5	a6
Earliest	0	0	3	2	2	7	5
Latest	1	0	4	6	2	8	5
Slack	1	**0**	1	4	**0**	1	**0**

Table 18.18: Slacks of the exam question.

The critical activities are a1, a4, and a6; the critical path is v0-v2-v4-v5, of length 2 + 3 + 5 = 10. All the values were also computed in a program with the same routines and agreed with those found by hand. The slack of a3 (4) is the largest: v2 occurs on day 2, while v3 can be waited for until day 8; there is a gap of 4 days in front of the 2-day a3.

18.9 Complexity summary

Table 18.19 gives all the algorithms covered in the chapter, the data structures they use, and their running times. *n* is the number of vertices and *e* the number of edges.

Algorithm	Data structure	Time	Extra memory
Brute force (single target)	all paths	$O(n^n)$	$O(n)$
Dijkstra (basic)	cost matrix + arrays	$O(n^2)$	$O(n)$
Dijkstra (improved)	min-heap + adjacency list	$O((n + e) \log n)$	$O(n)$
Printing the path	predecessor array	$O(\text{path length})$	$O(\text{path length})$
Bellman-Ford	edge list	$O(n \cdot e)$	$O(n)$
Floyd-Warshall	$n \times n$ matrix	$O(n^3)$	$O(n^2)$
Topological order	counted list + embedded stack	$O(n + e)$	$O(1)$
Earliest / latest time, slack	topological order + reverse list	$O(n + e)$	$O(n + e)$

Table 18.19: Complexity of the chapter's algorithms.

The extra-memory column shows the space allocated apart from the input itself. The extra memory of topological order is constant, because the stack is embedded in the header array. The time computations, however, need the reverse list and two arrays. All bounds in the table are for the worst

case. For basic Dijkstra and Floyd-Warshall the bound is also the best case: these routines always run the same number of rounds regardless of the input.

18.10 Common mistakes

1. **Using Dijkstra on a graph with negative edges.** The algorithm raises no error; it silently produces a wrong result. If there are negative weights, use Bellman-Ford, or relaxation in topological order for an acyclic graph.
2. **Choosing `INT_MAX` for infinity.** The sum `dist[u] + cost[u][w]` overflows; signed integer overflow is undefined behavior, and the result is often a negative number. Choose a constant that is larger than the real paths but whose double still fits in an `int`.
3. **Taking the smallest for the earliest time and the largest for the latest time.** The correct rule is the reverse: the earliest time waits for the largest of the entering activities, and the latest time must meet the smallest of the leaving activities.
4. **Computing the latest time before the successors are final.** The backward pass must be done in a topological order of the reversed graph, or in the reverse of the forward order. In an arbitrary order, the value of an event whose successor has not yet been processed is mistaken for final.
5. **Starting the topological order from an arbitrary vertex.** The earliest time of a vertex whose predecessors have not been processed is only a lower bound. A value becomes final only when the vertex is taken from the stack and written.
6. **Trying to build the path forward.** When leaving the source, it is not known which edge to choose. The predecessor array is read backward from the target and written in the right order by reversing with recursion or a stack.
7. **Printing a path to an unreachable vertex.** If `dist[t] >= INF`, then `pred[t]` is meaningless; this condition must be checked first.
8. **Calling `topological_order` a second time on the same structure.** The counter fields have turned into stack links; the counters must be recomputed.

18.11 Exercises and solutions

18.11.1 A different source

Question. What distances would `shortest_path` find if the source in the six-vertex example graph were 2?

Solution. Initially the row of 2 is copied: 20 for 0, 15 for 3, 1000 for the rest. In the first round 3 is selected (15); through 3 we get 15 + 20 = 35 for 1 and 15 + 35 = 50 for 4. In the second round 0 is selected (20); no value is shortened through 0 (70 for 1, 65 for 4). In the third round 1 is selected

(35); through 1 we get 35 + 10 = 45 for 4, which replaces 50. In the fourth round 4 is selected. Result: *d(2→0) = 20*, *d(2→3) = 15*, *d(2→1) = 35* (through 3), *d(2→4) = 45* (through 3 and 1); there is no path to 5.

18.11.2 Number of rounds of the outer loop

Question. Why does the outer loop of `shortest_path` run *n − 2* rounds? What would happen if it ran *n − 1* rounds?

Solution. The source is final at the start; after *n − 2* rounds, *n − 1* vertices are marked. For the one remaining vertex, all other vertices are in *S*, and its `dist` value is by definition the shortest length. If one more round were run, this vertex would be selected and marked, but since no unmarked vertex would be left to relax, no value would change. The extra round is harmless but unnecessary.

18.11.3 If an edge weight changes

Question. In the step-by-step seven-vertex example, what would be the order of joining *S* if the edge *v3 → v6* were 15 instead of 2?

Solution. The first four selections do not change: v1 (10), v2 (20), then v3 and v4 (both 30; on a tie the smaller index goes first: v3, v4). When v3 joins, the value of v6 is this time 30 + 15 = 45. When v4 joins, v5 drops to 40. Since 40 < 45, v5 is selected before v6. There is no edge from v5 to v6; v6 stays at 45. The order is v1, v2, v3, v4, v5, v6. The program gave the same order and the result `dist = 0 10 20 30 30 40 45`.

18.11.4 A queue instead of a stack

Question. If a queue were used instead of a stack in the topological-order routine, what would the output be in the v0…v5 example?

Solution. `0 1 2 3 5 4`. When v0 is taken, v1, v2, v3 enter the queue in this order and leave in the same order. When v1 is taken, the counter of v4 drops to 2; when v2 is taken, to 1; the counter of v5 drops to 1. When v3 is taken, the counter of v5 and then that of v4 drops to 0, because v5 comes before v4 in the list of v3. The queue version printed this output in the program too.

18.11.5 If an activity takes longer

Question. In the AOE example, what would the project duration and critical path be if a8 took 8 days instead of 4?

Solution. v7 is reached through v5 at 7 + 8 = 15, which is greater than the 14 coming through v4. `earliest[7] = 15`, and for v8 we get 15 + 4 = 19, so the project stretches to 19 days. The critical path is now v0-v3-v5-v7-v8 (5 + 2 + 8 + 4 = 19). The old critical paths took 18 days, so they are no

longer critical; the program found a slack of 1 day for a0, a3, a6, a7, and a9. The slack of a8 was 3 days; a non-critical activity that grows by more than its slack (4 days) changes the critical path. This is why the computation must be redone after every change.

18.11.6 A late change in Floyd-Warshall

Question. Why did `A[1][0]` change in round $k = 2$ of the Floyd-Warshall example?

Solution. The shortest path from 1 to 0 is $1 \to 2 \to 0$: $2 + 3 = 5$, while the direct edge is 6. This path uses 2 as an intermediate vertex. Since A^k allows only intermediate vertices among $0 \ldots k$, using 2 becomes possible only in round $k = 2$.

18.12 Summary

- The shortest-path problem is posed on a weighted directed graph; the length of a path is the sum of its edge costs. It has single-source single-target, single-source all-targets, and all-pairs versions.
- Brute force tries all simple paths. In a complete graph the number of paths is $1 + (n-2) + \cdots + (n-2)!$; even for a single target it needs work of the order of $O(n^n)$.
- Dijkstra's algorithm is greedy: it finalizes the vertex outside S with the smallest distance, then relaxes through it. It requires $w \geq 0$. It takes $O(n^2)$ with a matrix and $O((n + e) \log n)$ with a min-heap and adjacency lists.
- A constant that does not cause overflow is chosen for infinity. `min_vertex` returns −1 only when no unmarked vertex is left.
- The predecessor array alone suffices: shortest paths form a tree. The path is read backward from the target and written in the right order with recursion or a stack.
- Dijkstra goes wrong on a negative edge. Bellman-Ford relaxes all edges for $n - 1$ rounds and catches a negative cycle in round n: $O(n \cdot e)$. Floyd-Warshall releases the intermediate vertices one by one and finds all pairs in $O(n^3)$.
- An activity-on-vertex network shows the precedence relation and must be acyclic. A topological order is found by deleting predecessor-free vertices one at a time. With counted headers and an embedded stack it takes $O(n + e)$; the same routine also catches a cycle.
- In an activity-on-edge network the edges are activities and the vertices are events. The project duration is the longest path from start to finish: the critical path.
- Earliest times are found in topological order by taking the largest value (forward pass), and latest times on the reversed graph by taking the smallest value (backward pass). Slack is the difference between the latest and earliest start times. The activities with slack 0 form the critical path.

18.13 Key terms

The terms introduced or used in this chapter:

weighted graph shortest path brute force complete graph greedy method Dijkstra's algorithm relaxation predecessor successor priority queue heap Fibonacci heap negative cycle Bellman-Ford algorithm all-pairs shortest paths Floyd-Warshall algorithm activity network activity-on-vertex (AOV) network activity-on-edge (AOE) network directed acyclic graph topological order in-degree out-degree dummy activity critical path slack

18.14 Try It Yourself

1. In the six-vertex example graph, if 5 is taken as the source, in what order does Dijkstra finalize the vertices, and what are the distances?

   Answer

   The only edge from 5 goes to 3 (3). The order is 3 (3), 1 (23), 4 (33), 2 (38); 0 is the last remaining vertex, and its value is 58 through 2. The distances to 0 through 4 are 58, 23, 38, 3, 33, respectively. Row 5 of Floyd-Warshall gives the same values.

2. When `shortest_path` finishes, `pred = {0, 3, 0, 2, 0, 0}`. How many recursive calls does `print_path(pred, 0, 3)` make, and what does it print?

   Answer

   Three calls are made, including the first: `t` = 3, 2, 0. The output is `0 -> 2 -> 3`; the cost is 10 + 15 = 25.

3. In a Dijkstra code that uses `INT_MAX` as infinity, what can happen if an unreachable `u` is selected and `dist[u] + cost[u][w]` is computed?

   Answer

   The sum does not fit in an `int`; signed overflow is undefined behavior. In practice the result often wraps to a negative number, appears smaller than `dist[w]`, and a nonexistent "short path" is written.

4. In the four-vertex negative example, how many rounds does Bellman-Ford run and how many

relaxation attempts does it make in total? After which round does no value change?

Answer

$n - 1 = 3$ rounds with 4 edges each: 12 attempts; in round n another 4 edges are checked. The last change happens in the second round (`d[3]` from 6 to 4); the third round changes nothing.

5. What happens if the order of the loops in the Floyd-Warshall code is changed to `i`, `j`, `k` (with `k` innermost)?

Answer

The algorithm loses its correctness. The values `A[i][k]` and `A[k][j]` used in computing `A[i][j]` may have been taken before all the intermediate vertices were taken into account. The definition of A^k is satisfied when `k` is outermost.

6. In the v0…v5 topological-order example, what would `topological_order` print if the list of v3 were v4, v5 instead of v5, v4?

Answer

`0 3 2 5 1 4`; the output does not change. While v3 is processed, the counter of v4 drops to 2 and that of v5 to 1; since neither reaches zero, the order of entering the stack is not affected. List order changes only the order of vertices whose counters drop to zero at the same time.

7. In the AOE example, what would the project duration and critical path be if a6 took 10 days instead of 9?

Answer

v6 occurs at 7 + 10 = 17 at the earliest and v8 at 17 + 2 = 19; the project takes 19 days. The only critical path is v0-v1-v4-v6-v8. The slack of a7 and a10 becomes 1, because the path through v7 takes 18 days.

8. In the AOE example, what are the total float and the free float of a4? What does it mean that the two are equal?

Answer

The total float is 6 − 4 = 2, and the free float is *earliest[4] − earliest[2] − 1 = 7 − 4 − 1 = 2*. That they are equal shows that a4 can use its two days of allowance without affecting the earliest time of any successor; the allowance is not shared with another activity.

18.15 Further reading

- E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2nd ed., Sections 6.4 and 6.5. The example graphs of this chapter, the array version of Dijkstra's algorithm, topological order with an embedded stack, and the earliest/latest time computations follow the presentation there.
- T. H. Cormen et al., *Introduction to Algorithms*, 4th ed., Sections 20.4, 22, and 23. They give in detail the DFS-based topological sort, the correctness proofs of Dijkstra and Bellman-Ford, and the dynamic-programming analysis of Floyd-Warshall.
- R. Sedgewick, *Algorithms in C, Part 5: Graph Algorithms*, 3rd ed., Chapter 21. Shortest-path algorithms are treated with C code and experiments on large graphs.

18.16 References

1. E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2nd ed., Silicon Press, 2008. §6.4 Shortest Paths and Transitive Closure, §6.5 Activity Networks.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022. Section 20.4 (Topological Sort), Chapters 22 (Single-Source Shortest Paths) and 23 (All-Pairs Shortest Paths).
3. R. Sedgewick, *Algorithms in C, Part 5: Graph Algorithms*, 3rd ed., Addison-Wesley, 2002. Chapter 21 (Shortest Paths).

PART V

Sorting

Chapters 19–20

Basic Sorting Algorithms

Sorting is one of the most common things a computer does. This note first sets up the vocabulary of sorting: record, key, internal and external sorting, stability, and in-place sorting. It then writes the three basic algorithms, selection sort, insertion sort, and bubble sort, in C and traces them step by step on small arrays. For each one it computes the number of comparisons and moves in the best, worst, and average case. The average case of insertion sort is derived as an expected value over all permutations. The notion of an inversion ties the three algorithms to a single measure. Counting sort and bucket sort make no comparisons and run in linear time when the range or distribution of the data is known. Shell sort uses gapped insertion sort to destroy inversions quickly. The note ends with a theorem proved by a decision tree: no comparison-based sorting algorithm can use fewer than $\lceil \lg n! \rceil$ comparisons in the worst case.

LEARNING OUTCOMES

Concepts
Distinguish record, key, internal and external sorting, stability, and in-place sorting.

Three basic algorithms
Write selection sort, insertion sort, and bubble sort in C and trace an array step by step.

Analysis
Compute comparison and move counts for the best, worst, and average case, and explain why the average case is not the mean of the best and worst cases.

Sorting without comparisons
Explain when counting sort and bucket sort run in O(n) time, and show why counting sort is stable.

Shell sort
Use inversions to show why gapped insertion sort is faster than plain insertion sort.

Lower bound
Prove the Ω(n lg n) lower bound for comparison sorting with a decision tree.

19.1 Sorting concepts

Names in a phone book, the results list of an exam, the pages returned by a search engine, a shop's products ordered by price: all of these are sorted data. Sorted data offers two big conveniences. First, searching gets faster. In a sorted array, binary search reaches its answer in $O(\lg n)$ steps, while in an unsorted array every element has to be examined. Second, many questions answer themselves on sorted data. The smallest and largest values sit at the two ends of the array. Equal values land next to each other, so finding duplicates takes a single pass. The common elements of two lists can be found by walking both sorted lists at once.

Sorting algorithms are also the best training ground for algorithm analysis. The problem is easy to understand and the solutions are short, yet their costs differ enormously. One algorithm sorts the same thousand numbers with half a million comparisons, another with fifteen thousand. This note first builds a common vocabulary. It then studies three basic algorithms in detail, followed by two algorithms that make no comparisons and by Shell sort. It closes with the lower bound that every comparison sort must obey.

19.1.1 Record, field, and key

What gets sorted is often not individual numbers. A student-records program keeps a student's number, first name, surname, and grade average together. Each of these items is called a **field**, and the whole set of fields belonging to one student is a **record**. In C, a record is naturally represented by a `struct`:

```
typedef struct {
    int    no;          /* key */
    char   name[20];
    char   surname[20];
    double avg;
} student;
```

A collection of records held in memory is called a **list**; when the records are kept outside memory, for example on disk, the collection is called a **file**. Sorting is always done on one field or on several of them.

Definition 19.1

A field, or a group of fields, that distinguishes a record from all others, that is, identifies it uniquely, is called a **key**. **Sorting** is rearranging a list of n records so that their keys are in increasing (or decreasing) order.

Figure 19.1 shows three student records. The student number is the key: two students may have the same first name and surname, but not the same number. Name, surname, and average are data fields. Sorting compares only the keys, but all the fields of a record move together with the key. This distinction creates two costs: the cost of comparing keys and the cost of moving records. If the records are large, the second can dominate.

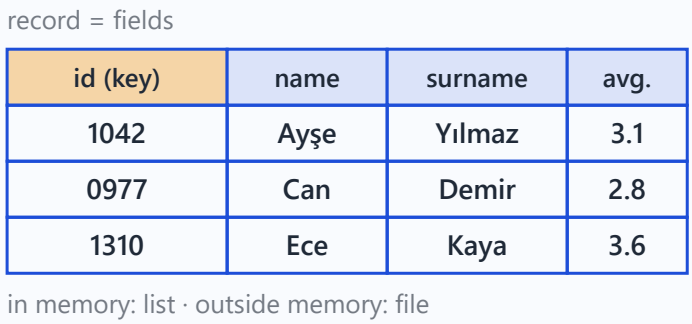


Figure 19.1: Three student records. The "id" field, with the orange header, is the key; name, surname, and average are data fields.

In real programs the key does not have to be unique. Students can also be sorted by grade average, and two students may have the same average. In that case the word "key" means the field the sorting is based on. What order records with equal keys keep after sorting is a matter of stability.

Sorting is also divided in two by where the data lives. In **internal sorting** all the data is in main memory and any element is reached in constant time. In **external sorting** the data does not fit in memory, and auxiliary storage such as a disk or tape is used. Reading one block from disk takes longer than thousands of comparisons in memory, so in external sorting the cost is determined by the number of disk accesses, not the number of comparisons. This note is about internal sorting. Still, its ideas are also the basis of external sorting: external sorting algorithms sort the pieces that fit in memory with an internal sort, then merge the sorted pieces.

19.1.2 What do we measure?

Measuring the running time of a sorting algorithm directly in seconds is misleading, because the time varies with the machine, the compiler, and the input. Instead we count the basic operations the algorithm performs. Sorting has two basic operations:

- **Comparison.** The order of two keys is queried, as in `a[j] < a[min]`. Comparison-based algorithms learn about the input only through such queries.
- **Move.** An element is written to another place in the array. A swap consists of three assignments; the shifting in insertion sort is a single assignment. In this note, "move" covers both; when it matters, we say which one is counted.

An algorithm's running time is proportional to the sum of these two counts. Which one matters more depends on the data. If the keys are long strings, comparison is expensive. If the records are large, moving is expensive. On flash memory, every write wears the cell a little, so reducing the number of writes is valuable in its own right. Counts are given for three cases. The **best case** is the input on which the algorithm does the least work, the **worst case** the input on which it does the most. The **average case** is the expected work under the assumption that all possible inputs are equally likely. This definition of the average case matters, and it is worked out in detail for insertion sort.

19.1.3 Stability

Definition 19.2

A sorting algorithm is **stable** if records with equal keys keep, after sorting, the relative order they had before. Precisely: if $i < j$ in the input and the keys of $a[i]$ and $a[j]$ are equal, then $a[i]$ also comes before $a[j]$ in the output.

Figure 19.2 illustrates the definition with a small example. The input has five records; the number is the key, and the letter is added only to tell the records apart. 3a and 3c have the same key, and 3a comes first in the input. A stable algorithm also puts 3a before 3c in the output: *1b 1e 2d 3a 3c*. Selection sort, however, produces *1b 1e 2d 3c 3a*. The output is correctly sorted by key, but 3c has moved ahead. How selection sort arrives at this result is shown step by step once the algorithm is introduced.

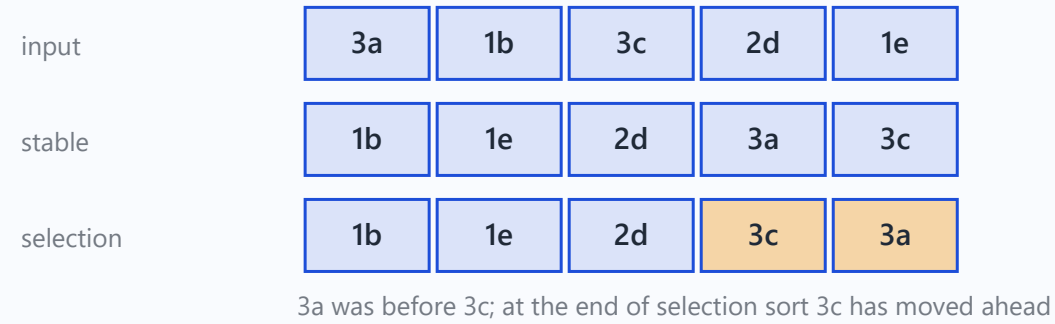


Figure 19.2: Stability. In the input, 3a is before 3c. A stable sort preserves this order; selection sort moves 3c ahead.

When bare numbers are sorted, stability does not matter: there is no way to tell two 3s apart. Stability becomes important when the records have fields other than the key. The best-known use is sorting on several fields.

Example 19.1

A class list is to be sorted by surname, and students with the same surname by first name. The list consists of (first name, surname) pairs: (Mehmet, Kaya), (Ali, Yılmaz), (Zeynep, Kaya), (Ayşe, Yılmaz), (Can, Demir).

First the whole list is sorted by first name: (Ali, Yılmaz), (Ayşe, Yılmaz), (Can, Demir), (Mehmet, Kaya), (Zeynep, Kaya). Then the same list is sorted by surname with a *stable* algorithm. Demir is alone and moves to the front. The relative order of the two Kayas is preserved: after the first step Mehmet was before Zeynep, and he stays there. For the two Yılmazes, Ali also stays before Ayşe. The result: (Can, Demir), (Mehmet, Kaya), (Zeynep, Kaya), (Ali, Yılmaz), (Ayşe, Yılmaz). The surnames are in order, and those with the same surname are in first-name order.

If the second sort were unstable, the work of the first sort could be wasted: Zeynep Kaya could move ahead of Mehmet Kaya.

Note the order in the example: first the *secondary* field (first name) is sorted, then the *primary* field (surname). The last sort dominates, and the earlier ones remain to break ties. Radix sort applies the same idea to the digits of numbers. Numbers are sorted first by the ones digit, then by the tens digit, and finally by the most significant digit; a stable algorithm is used in each pass. That is exactly why the algorithm used in each pass must be stable.

19.1.4 In-place sorting

Definition 19.3

A sorting algorithm that uses only a constant amount of extra memory beyond the array holding the input, that is, *O(1)*, is called an **in-place** algorithm. Some sources also count *O(lg n)* extra memory as in-place, to allow for a recursion stack.

Selection, insertion, and bubble sort are in place: apart from a few loop variables and one temporary variable for swapping, they need no memory. Counting sort is not in place; it writes its output to a separate array `b` and uses an array `c` for the counters. Bucket sort also needs *O(n)* extra memory for the buckets and nodes. When memory is scarce, for example in an embedded controller, an in-place algorithm may therefore be the only option.

19.1.5 Shared infrastructure: the `less` and `SWAP` macros

The comparison-based algorithms in this note use the same two building blocks: `less`, which compares two elements, and `SWAP`, which swaps two elements. Both are macros of the C preprocessor. A macro is an abbreviation that is replaced as text before compilation; it has no

function-call cost.

```
#define N 15
#define less(A, B) ((A) < (B))
#define SWAP(A, B) do { int t_ = (A); (A) = (B); (B) = t_; } while (0)
```

- **Line 1.** `N` is the number of elements the test program will sort.
- **Line 2.** `less(A, B)` yields 1 if $A < B$ and 0 otherwise. Parenthesizing the parameters and the whole expression is deliberate. A macro is expanded as text. With an unparenthesized definition, the call `less(x & 1, y)` would turn into the text `x & 1 < y`. Since the `<` operator is applied before `&` in C, this expression would be evaluated as `x & (1 < y)`. When the algorithms do their comparisons through this macro, changing the comparison criterion is a one-line job. For example, to sort from largest to smallest, it is enough to write `>` instead of `<` in the definition.
- **Line 3.** `SWAP(A, B)` saves the value of `A` in the temporary `t_`, writes `B` into `A`, and writes the old `A` into `B`. Three assignments are made. This version takes two arguments and uses `int` as the element type; for other element types, change the `int` in the definition.

Wrapping the body of the macro in `do { … } while (0)` looks odd at first. The loop runs once and nothing repeats. Its purpose is to make the macro a single statement syntactically. Let us see what would happen if the body were written with braces alone:

```
#define SWAP(A, B) { int t = A; A = B; B = t; }

if (a[0] > a[1])
    SWAP(a[0], a[1]);
else
    puts("sorted");
```

The preprocessor turns the line `SWAP(a[0], a[1]);` into the text `{ int t = a[0]; a[0] = a[1]; a[1] = t; };`. The semicolon after the block is an empty statement and ends the `if` statement. The `else` that follows is left dangling, and the compiler reports an error. With the `do { … } while (0)` form, the trailing semicolon closes the `while (0)`, and the `if`–`else` structure stays intact. This idiom is established practice in C for macros made of several statements.

In the algorithms the two macros often appear together: `if (less(A, B)) SWAP(A, B);`. This line swaps the two elements if `A < B`. The pattern is called **compare-exchange**. One has to watch its direction: if `A` is *to the right of* `B` in the array, this line moves the smaller element to the left. The line `if (less(a[j], a[j - 1])) SWAP(a[j], a[j - 1]);` in bubble sort is exactly that: if the right element `a[j]` is smaller than the left one, they trade places.

Common mistake

Passing an argument with side effects to a macro. A macro is not a function; arguments are copied as text without being evaluated. In the call `SWAP(a[i++], a[j])`, the text `a[i++]` appears twice in the body, so `i` is incremented twice and the swap is done with the wrong elements. With `a = {10, 20, 30, 40}`, `i = 0`, and `j = 3`, after this call `i` is 2 and the array becomes `{10, 40, 30, 10}`: 20 is lost and 10 is written twice. If a macro whose body contains its argument twice is given `f(x)`, `f` is also called twice. Macro arguments should always be simple expressions without side effects.

The main program used to test the algorithms is also short:

```
int main(void) {
    int *a = malloc(N * sizeof *a);
    if (a == NULL) return 1;
    for (int i = 0; i < N; i++)
        a[i] = rand() % 50;
    selection(a, 0, N - 1);   /* call the sorting routine */
    for (int i = 0; i < N; i++) printf("%d ", a[i]);
    free(a);
    return 0;
}
```

- **Lines 2–3.** An array of `N` integers is allocated from the heap. `sizeof *a` is the size of the element `a` points to, that is, `sizeof(int)`. This form stays correct automatically if the array's type changes. If `malloc` finds no memory, it returns `NULL`; the program then exits with code 1. Without the check, writing `a[i]` would try to write to the `NULL` address and the program would crash.
- **Lines 4–5.** The array is filled with random numbers between 0 and 49. `rand()` produces the same array on every run, because no seed (`srand`) is given. This is a convenience in testing: when a bug is found, the same input can be reproduced.
- **Line 6.** The sorting routine is called. All routines sort the part `a[L..R]` of the array; to sort the whole array, pass `L = 0` and `R = N - 1`. This interface also makes it possible to sort only a part of an array. Algorithms such as quicksort hand small parts over to insertion sort through this interface.
- **Lines 7–8.** The result is printed and the memory is released. If `free` is forgotten, nothing happens in a small program. But if the same code is called repeatedly in a long-running program, memory leaks accumulate.

The program is compiled with the `stdio.h` and `stdlib.h` headers. The numbers `rand()` produces vary with the C library. On Linux (glibc) the output is `9 12 13 15 21 27 27 33 35 36 36 40 42 43 49`; in another library, for example on Windows, the numbers are different.

19.2 Selection sort

19.2.1 Idea and code

Selection sort is perhaps the most natural way to sort. Someone putting a stack of exam papers on a desk into order of student number first looks for the paper with the smallest number and puts it on top. Then among the rest, they look for the smallest and put it second. This goes on until the papers run out. The algorithm takes three sentences to state:

1. Find the element with the smallest value in the list.
2. Swap it with the element in the first position.
3. Repeat the same steps for the remaining sublist, starting from the second element.

The implementation in C:

```
void selection(int a[], int L, int R) {
    for (int i = L; i < R; i++) {
        int min = i;
        for (int j = i + 1; j <= R; j++)
            if (less(a[j], a[min]))
                min = j;
        SWAP(a[i], a[min]);
    }
}
```

- **Line 2.** The outer loop walks `i` over the position where the next smallest element will settle. `i` goes up to `R - 1`, not to `R`. Once $n - 1$ elements are in place, the last one is necessarily in the right place: the one remaining element is the largest.
- **Line 3.** `min` holds the *index* of the smallest element seen so far. Initially the candidate is the first element of the remaining sublist, `a[i]`.
- **Lines 4–6.** The inner loop scans the rest of the remaining sublist, `a[i+1..R]`. When a smaller element is seen, `min` is updated to its index. The comparison is made with the `less` macro, that is, with strict "less than": an equal element does not change `min`.
- **Line 7.** The smallest element is swapped with `a[i]`. If `min == i`, the swap exchanges the element with itself; the array does not change, but the three assignments are still made.

The correctness of the algorithm is shown with a loop invariant. An invariant is a property that stays true at the end of every iteration of the loop.

> At the end of each outer pass, *a[L..i]* is sorted and contains the $i - L + 1$ smallest elements of the array.

After the first pass, *a[L]* holds the minimum of the whole array; a one-element part is sorted. If the invariant holds for $i - 1$ at the start of a pass, the pass finds the smallest of the remaining elements and puts it in *a[i]*. This element is not smaller than any element in *a[L..i−1]* (those were chosen earlier) and is not larger than any of the remaining ones. The invariant therefore holds for i too. When the loop ends with $i = R - 1$, *a[L..R−1]* is sorted and carries the $n - 1$ smallest elements; the remaining *a[R]* is the largest. The array is sorted.

A notable property of selection sort is that it never looks at the state of the input. To select the minimum, the remaining sublist has to be scanned to the end in every pass. Even if the array is sorted from the start, the algorithm does not notice, because it learns that "no smaller element remains" only when it finishes the scan.

19.2.2 Step-by-step trace

The call `selection(a, 0, 4)` is traced on the array *a = [29, 10, 14, 37, 13]*. Table 19.1 gives each comparison on its own row; Figure 19.3 shows the swap at the end of each pass.

`i`	`j`	**Comparison**	`min`	**Array**
0	1	10 < 29 true	1	[29, 10, 14, 37, 13]
0	2	14 < 10 false	1	
0	3	37 < 10 false	1	
0	4	13 < 10 false	1	
0	—	`SWAP(a[0], a[1])`	1	[**10**, 29, 14, 37, 13]
1	2	14 < 29 true	2	
1	3	37 < 14 false	2	
1	4	13 < 14 true	4	
1	—	`SWAP(a[1], a[4])`	4	[10, **13**, 14, 37, 29]
2	3	37 < 14 false	2	
2	4	29 < 14 false	2	
2	—	`SWAP(a[2], a[2])` (with itself)	2	[10, 13, **14**, 37, 29]
3	4	29 < 37 true	4	
3	—	`SWAP(a[3], a[4])`	4	[10, 13, 14, **29**, **37**]

Table 19.1: Trace of `selection(a, 0, 4)`, *a = [29, 10, 14, 37, 13]*. Each row is a comparison or the swap at the end of a pass.

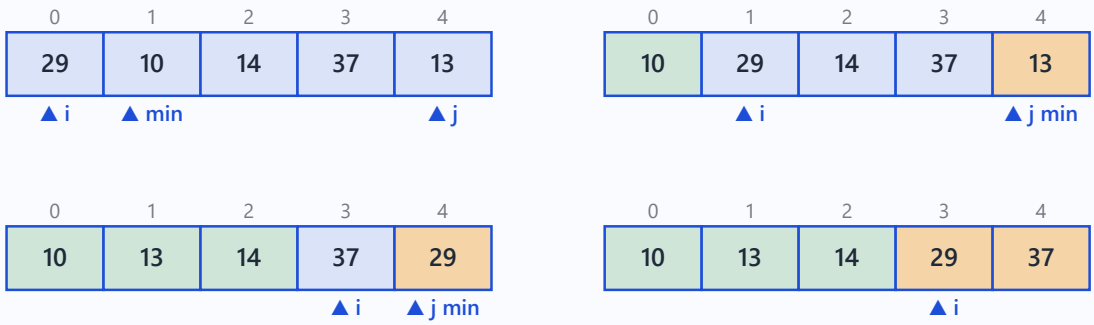


Figure 19.3: The end of each pass of selection sort, left to right and top to bottom. (1) The scan of pass 0 is done; the minimum is 10 (`min = 1`). (2) At the end of pass 1 the minimum is 13 (`min = 4`). (3) In pass 3, 29 was selected. (4) The sorted array after the last swap. Green boxes show elements that are in place, orange boxes the current minimum candidate or the elements just swapped; arrows point to the indices `i` , `j` , and `min` .

Three details stand out in the trace. First, in pass 0 `min` changed only once, because 13 is also larger than 10. In pass 1 it changed twice: first to 14, then to 13. How many updates occur depends on the input, but the number of comparisons does not. Second, in pass 2, 14 was already in place and the swap was with itself. Third, the array was not yet sorted before pass 3; the last swap put two elements in place at once. In total there were 4 + 3 + 2 + 1 = 10 comparisons and 4 calls to swap.

19.2.3 Analysis

In pass i, the inner loop runs for $j = i + 1, \ldots, R$. In an array of n elements ($L = 0$, $R = n - 1$), the first pass makes $n - 1$ comparisons, the second $n - 2$, and the last 1:

$$C(n) = (n - 1) + (n - 2) + \ldots + 2 + 1 = n(n - 1)/2 \approx n^2/2$$

This total is the sum of the first $n - 1$ natural numbers. The familiar way to find it is to write the terms again in reverse order and add them column by column. Each column sums to n, and there are $n - 1$ columns. Twice the sum is $n(n - 1)$, so the sum itself is half of that.

The number of comparisons is independent of the input: whether the array is sorted, reverse sorted, or random, the inner loop runs the same number of times. For selection sort the best, average, and worst cases are the same, and all are $\Theta(n^2)$. The number of swaps is exactly $n - 1$: one swap per outer pass, and there is no pass for the last element. Some of them are swaps of an element with itself; guarding the code with `if (min != i)` avoids the needless swaps, but adds a comparison to every pass. The extra memory is only `i` , `j` , `min` , and the temporary variable of the swap: $O(1)$. The algorithm is in place.

n	Comparisons $n(n - 1)/2$	Swaps $n - 1$
5	10	4

n	Comparisons *n(n – 1)/2*	Swaps *n – 1*
10	45	9
100	4.950	99
1.000	499.500	999

Table 19.2: Comparison and swap counts of selection sort. The values are the same for every input.

Table 19.2 shows how differently the two counts grow. When n grows tenfold, the number of comparisons grows about a hundredfold, while the number of swaps grows only tenfold. If swaps are expensive, this difference makes selection sort valuable. When records are large, say a few hundred bytes each, every swap means three large copies. On flash memory every write wears the cell. In such cases the $n - 1$ swaps of selection sort are much cheaper than the moves of the other basic algorithms, which are of order n^2.

Selection sort is not stable. The cause is the long-distance swap. There can be many elements between `a[i]` and `a[min]`. `a[i]` jumps over them and may land behind an element with an equal key. On the input of Figure 19.2 (3a, 1b, 3c, 2d, 1e), this can be seen step by step:

- **Pass 0.** The minimum is 1b (`min = 1`). 3a and 1b are swapped: (1b, 3a, 3c, 2d, 1e).
- **Pass 1.** The smallest of the rest is 1e (`min = 4`). 3a and 1e are swapped: (1b, 1e, 3c, 2d, 3a). 3a jumped over 3c and 2d to the very end. Stability was broken at this step.
- **Pass 2.** The minimum is 2d. 3c and 2d are swapped: (1b, 1e, 2d, 3c, 3a).
- **Pass 3.** `less(3a, 3c)` is false; `min` does not change and the swap is with itself. The result is (1b, 1e, 2d, 3c, 3a).

On this input the order of 1b and 1e is preserved, but that is a coincidence. Stability is a property that must hold for all inputs; a single counterexample makes an algorithm unstable. The smallest counterexample has three elements and is taken up in the exercises.

19.3 Insertion sort

19.3.1 Idea and code

Insertion sort is the way someone playing cards arranges the cards in their hand. The cards are picked up from the table one by one. The cards in hand are sorted at every moment; the new card is inserted at the right place among them. To do this, the cards larger than the new one are pushed one place to the right and the new card is put into the gap that opens.

The same is done in an array. A sorted **prefix** grows at the left of the array. Initially the prefix is just the first element, and a single element is always sorted. In each pass, the element right after the prefix is taken and inserted into its place within the prefix. At the end of the pass the prefix is one element

longer. The first of the two versions is this:

```
void insertion(int a[], int L, int R) {
    for (int i = L + 1; i <= R; i++) {
        int v = a[i];      // element to insert
        int j = i - 1;     // a[L..i-1] is sorted
        while (j >= L && a[j] > v) {
            a[j + 1] = a[j]; // shift right
            j--;
        }
        a[j + 1] = v;
    }
}
```

- **Line 2.** `i` is the index of the element to be inserted and starts at `L + 1`. At this moment `a[L..i-1]` is sorted.
- **Line 3.** The value to be inserted is saved in `v`. This step is mandatory: the first shift overwrites `a[i]` and the value would be lost.
- **Line 4.** `j` starts at the right end of the prefix and moves left.
- **Line 5.** The loop runs as long as two conditions are both true: `j` is inside the subarray (`j >= L`) and `a[j]` is greater than `v`. C's `&&` operator short-circuits: if the first condition is false, the second is never evaluated. When `j` drops to $L - 1$, `a[j]` is not read; if it were, the access would go outside the array.
- **Lines 6–7.** The element greater than `v` is shifted one place to the right and `j` is decremented. A shift is a single assignment, not a swap.
- **Line 9.** When the loop ends, either `a[j]` is not greater than `v` or `j` is to the left of the subarray. In both cases the place of `v` is `j + 1`.

The loop invariant this time is: at the start of each pass of the outer loop, *a[L..i−1]* is the sorted version of the elements that were in those positions in the input. The difference from selection sort's invariant is important. In insertion sort the prefix is sorted, but it need not contain the smallest elements of the array: a small element arriving later can enter at the very front of the prefix. The algorithm does not know for certain the final place of any element until the last pass.

Common mistake

Writing the loop condition as `j >= 0`. When the whole array is sorted, $L = 0$, so the bug is invisible. When the routine is called on a subarray, for example `insertion(a, 2, 5)`, `j` passes to the left of `L` and goes down to `a[0]` and `a[1]`. These elements do not belong to the part being sorted; the shifting corrupts them. The correct bound is `j >= L`. For *a = [9, 8, 7, 1, 6, 5, 0]*, `insertion(a, 2, 5)` with the correct code gives *[9, 8, 1, 5, 6, 7, 0]*: only *a[2..5]* is sorted.

19.3.2 Step-by-step trace

The call `insertion(a, 0, 4)` is traced on *a = [7, 2, 9, 5, 1]*. Table 19.3 gives the comparisons and shifts of each pass and the array at the end of the pass.

`i`	`v`	Tests	Shifts	Comp.	End of pass
1	2	7 > 2 true; `j = -1`, loop ends	1	1	[**2**, 7, 9, 5, 1]
2	9	7 > 9 false	0	1	[2, 7, **9**, 5, 1]
3	5	9 > 5 true, 7 > 5 true, 2 > 5 false	2	3	[2, **5**, 7, 9, 1]
4	1	9, 7, 5, 2 > 1 true; `j = -1`, loop ends	4	4	[**1**, 2, 5, 7, 9]
Total			**7**	**9**	

Table 19.3: Trace of `insertion(a, 0, 4)`, *a = [7, 2, 9, 5, 1]*. The "Comp." column is the number of `a[j] > v` tests.

The most instructive pass is *i = 3*. Figure 19.4 shows the intermediate states of this pass. `v = 5` has been set aside. 9 is greater than 5 and shifts one place right; two 9s temporarily appear in the array. Then 7 shifts and two 7s appear. 2 is less than 5; the loop stops and 5 is written into `a[1]`. It is normal for the array to hold two copies of a value in the intermediate states. A shift copies an element to the right; the old place is filled in the next step either with another element or with `v`.

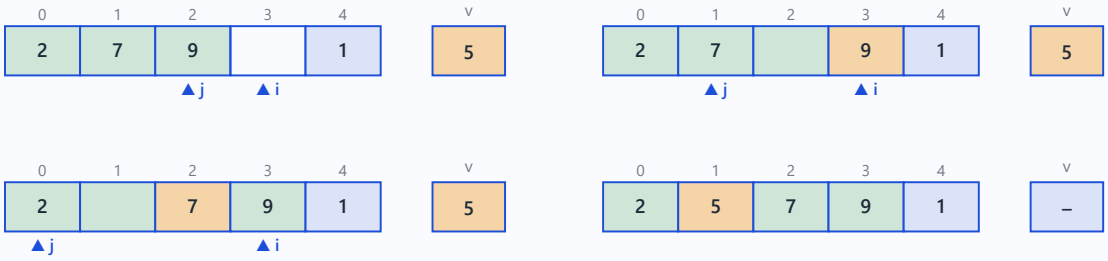


Figure 19.4: Pass *i = 3* of insertion sort, left to right and top to bottom. (1) `v = 5` is set aside. (2) 9 shifted one place right. (3) 7 shifted one place right. (4) *2 ≤ 5*, the loop stopped and 5 was written into `a[1]`. Green boxes show the sorted prefix.

The trace counted 7 shifts and 9 comparisons. The number of shifts is no coincidence: 7 is the number of inverted pairs in the array. This connection is proved separately together with the notion of an inversion. The number of comparisons is slightly more than the number of shifts. A pass ends either when `a[j] > v` turns out false, and this last test is a comparison without a shift; or when `j` runs off the beginning of the array, in which case there is no extra comparison. Here the passes *i = 2* and *i = 3* ended the first way, and the passes *i = 1* and *i = 4* the second way. The comparison count came to *7 + 2 = 9*.

19.3.3 The sentinel version: `insertion1`

Each step of the inner loop makes two tests: `j >= L` and `a[j] > v`. The first is useful only if `v` will go all the way to the beginning of the array. That happens only when `v` is the smallest of the elements seen so far. If the smallest element of the array is placed at `a[L]` from the start, no `v` can pass to its left; `a[L]` stops every loop by itself. Such an element is called a **sentinel**. The second version uses this idea:

```
void insertion1(int a[], int L, int R) {
    for (int i = R; i > L; i--) // sentinel
        if (a[i] < a[i-1]) SWAP(a[i], a[i-1]);
    for (int i = L + 2; i <= R; i++) {
        int v = a[i], j = i;
        while (less(v, a[j - 1])) {
            a[j] = a[j - 1];
            j--;
        }
        a[j] = v;
    }
}
```

- **Lines 2–3.** Preparation pass: neighbors are compared from the right end of the array to the left, and the right one is swapped if it is smaller. This single pass carries the smallest element to `a[L]` like a bubble. It makes $n - 1$ comparisons and at most $n - 1$ swaps.
- **Line 4.** The main loop starts at `L + 2`. After the preparation pass `a[L]` is the smallest, so `a[L..L+1]` is already sorted.
- **Lines 5–10.** In the inner loop only `less(v, a[j - 1])` is tested. When `j` drops to `L + 1`, `a[L]` cannot be greater than `v`, so the condition becomes false and the loop stops. It is impossible for `j` to pass to the left of `L`.

This version is faster for two small reasons. The inner loop makes one test fewer per step; when the loop runs thousands of times, the difference is measurable. Also, since the condition is a single comparison, the compiler can compile the loop more tightly. The price is the preparation pass's extra $n - 1$ comparisons. Table 19.4 traces the call `insertion1(a, 0, 3)` on $a = [4, 3, 1, 2]$.

Phase	Test	Action	Array
Sentinel, `i = 3`	2 < 1 false	no swap	[4, 3, 1, 2]
Sentinel, `i = 2`	1 < 3 true	swap	[4, 1, 3, 2]
Sentinel, `i = 1`	1 < 4 true	swap; sentinel ready	[**1**, 4, 3, 2]
`i = 2`, `v = 3`	3 < 4 true	4 shifts right, `j = 1`	[1, 4, 4, 2]
	3 < 1 false	`a[1] = 3`	[1, **3**, 4, 2]
`i = 3`, `v = 2`	2 < 4 true	4 shifts right, `j = 2`	[1, 3, 4, 4]
	2 < 3 true	3 shifts right, `j = 1`	[1, 3, 3, 4]

Phase	Test	Action	Array
	2 < 1 false (sentinel)	`a[1] = 2`	[1, **2**, 3, 4]

Table 19.4: Trace of `insertion1(a, 0, 3)`, *a = [4, 3, 1, 2]*.

In the last pass, `v = 2`, after shifting two elements, met `a[0] = 1` and stopped. It could stop safely without a bounds check, because the sentinel is less than or equal to every `v`. The preparation pass has a side effect too: not only the smallest element but also other small elements along the way move somewhat to the left. Here, as 1 passed ahead of 3 and 4, 3 was also pushed one place right. So in later passes the shifting work may be slightly reduced.

The sentinel idea is not specific to sorting. In linear search on an array, if the sought value is temporarily written at the end of the array, the loop does not have to ask "is the array finished?" at every step. In linked lists, a header node does the same job: even when the list is empty, the first node has a predecessor and the special case disappears.

19.3.4 Best and worst case

Unlike selection sort, the work of insertion sort depends heavily on the input. Figure 19.5 puts the two extreme cases side by side.

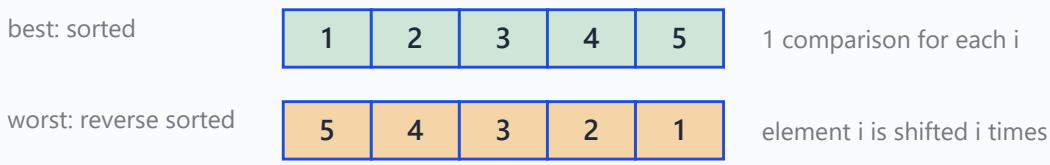


Figure 19.5: The best case of insertion sort is sorted input (a single comparison per element); the worst case is reverse-sorted input (every element passes the entire prefix).

Best case: sorted input. In every pass `v` is compared with the rightmost element of the prefix. That element is less than or equal to `v`; the loop never runs and `v` stays where it is. One comparison is made in each of the $n - 1$ passes, and no shifts. The total is $n - 1$ comparisons; the running time is $\Theta(n)$. No algorithm can do better: to be sure the array is sorted, every element must be looked at at least once.

Worst case: reverse-sorted input. Every new element is smaller than all those seen so far. In pass i, `v` is compared with all i elements of the prefix, all of them shift one place right, and `v` is written at the very front. The total is

$$1 + 2 + \ldots + (n - 1) = n(n - 1)/2$$

comparisons and the same number of shifts. The running time is $\Theta(n^2)$. For $n = 1{,}000$ that is 499,500 comparisons and 499,500 shifts.

Insertion sort is stable. The condition of the inner loop is a strict inequality: `a[j] > v`. When an element equal to `v` is met in the prefix, the condition becomes false and `v` is written to the *right* of that element. An element that comes later in the input stays later in the output. If the condition were `a[j] >= v`, `v` would jump over the equal elements and land to their left; the array would still be sorted, but stability would be lost and needless shifts would be made. The algorithm is in place: it uses no memory beyond `i`, `j`, and `v`.

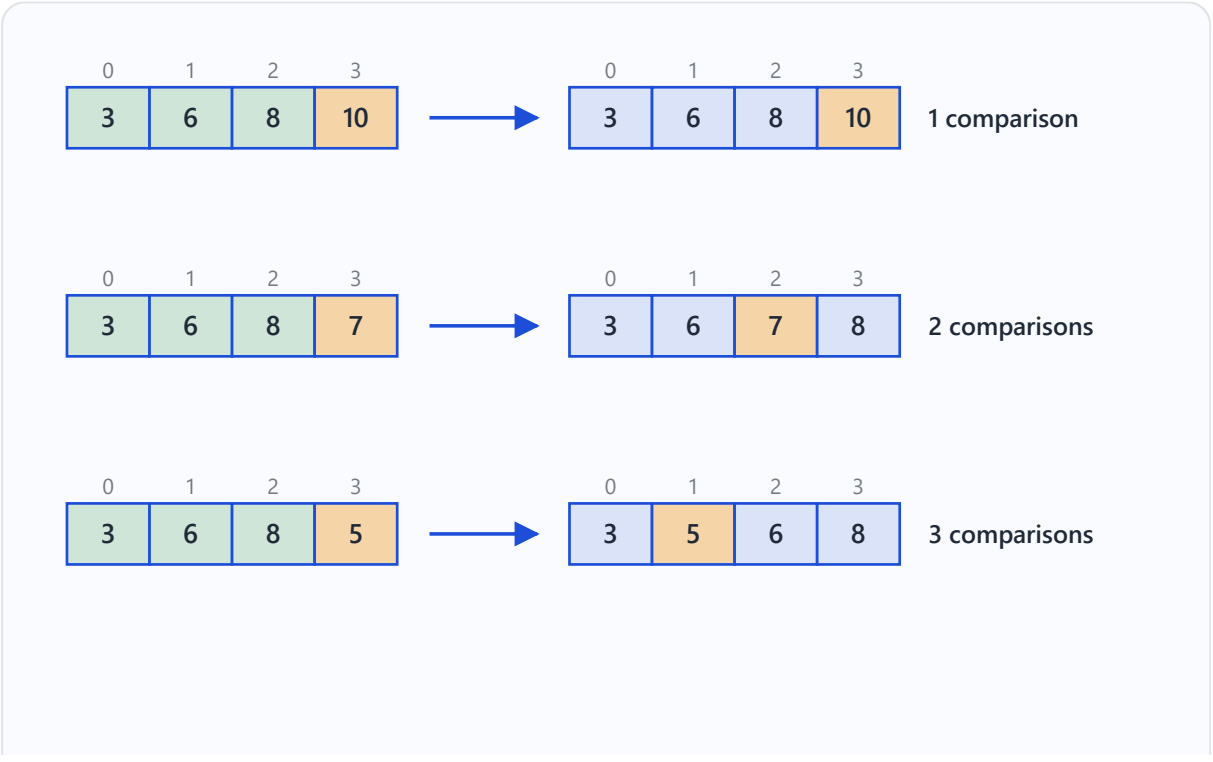
The real strength of insertion sort is "nearly sorted" data. If every element is a few positions from its final place, each pass ends in a few steps and the total work stays close to *n*. Applications that add an occasional new record to a sorted list, algorithms that make a final pass over a mostly sorted array, and small subarrays are in this situation. Practical implementations of quicksort switch to insertion sort when the parts fall below 10–20 elements.

19.3.5 Average case: an example

The best case is $n - 1$ comparisons, the worst case $n(n - 1)/2$. How many comparisons should we expect for a random input? The first idea that comes to mind is to take the mean of the two extremes. That is wrong. In an average-case analysis, all cases in the sample space must be observed and each weighted by its probability. The mean of the extremes accounts for only two cases.

First the average of a single pass is found. In pass j (this section uses j for the pass index), $a[0..j-1]$ is sorted and $a[j]$ is to be inserted. If the input is random, the rank of $a[j]$ among the j elements of the prefix is also random. The element is placed in any of the $j + 1$ positions with equal probability: at the right end of the prefix, one place to its left, …, at the very front.

Figure 19.6 shows these four cases for $j = 3$ on the sorted prefix $[3, 6, 8]$.



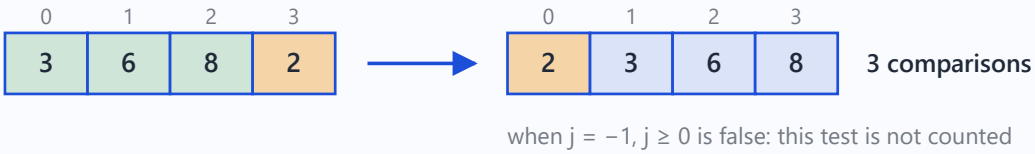


Figure 19.6: Four cases for *j = 3*. From top to bottom, the values 10, 7, 5, and 2 are inserted into the prefix *[3, 6, 8]*; the numbers of comparisons needed are 1, 2, 3, and 3 respectively.

- **Case 1,** `v = 10`**.** *8 > 10* is false. 1 comparison.
- **Case 2,** `v = 7`**.** *8 > 7* is true, *6 > 7* is false. 2 comparisons.
- **Case 3,** `v = 5`**.** *8 > 5* and *6 > 5* are true, *3 > 5* is false. 3 comparisons.
- **Case 4,** `v = 2`**.** *8 > 2*, *6 > 2*, *3 > 2* are true. Then `j = -1`, `j >= 0` turns out false, and because of short-circuiting `a[j] > v` is never tested. 3 comparisons.

The last two cases need the same number of comparisons. If the element goes to the very front, the last comparison does not give a "stop" answer; the bounds test ends the loop. That is why there is no case with 4 comparisons. The four cases are equally likely, and the expected number of comparisons is

$$(1 + 2 + 3 + 3) / 4 = 9/4 = 2{,}25$$

This number differs from 2, the mean of the best pass (1) and the worst pass (3). Here the difference is small, but the method is wrong and leads to large errors in other algorithms. For bubble sort with early exit, the mean of the extremes gives $\approx n^2/4$ comparisons; counting over all permutations shows that the true average is $\approx n^2/2$.

19.3.6 Average case: the general computation

To generalize the computation we need a notion from probability. A number whose value depends on chance is called a **random variable**. Let C_j be the number of comparisons made while inserting the jth element. Its value depends on the input, so it is a random variable. The **expected value** $E(C_j)$ of a random variable is found by multiplying each value it can take by its probability and summing. If all cases are equally likely, the expected value is the simple arithmetic mean.

In pass j there are $j + 1$ cases. If the element is to land at the kth position from the right ($k = 1, \ldots, j$), k comparisons are made. If it is to land at the very front, that is, in the $(j + 1)$th case, j comparisons are made. Thus

$$E(C_j) = (1 / (j + 1)) \cdot [1 + 2 + \ldots + j + j] = (1 / (j + 1)) \cdot [2j + \Sigma_{k=1}^{j-1} k]$$

The expression in the brackets is $j(j + 1)/2 + j$. Dividing by $j + 1$ gives

$$E(C_j) = j/2 + j/(j + 1) = j/2 + 1 - 1/(j + 1) = (j + 2)/2 - 1/(j + 1)$$

For $j = 3$ this is $5/2 - 1/4 = 2.25$, the value found by hand in the example. For $j = 4$ it is $3 - 1/5 = 2.8$: the comparison counts of the five cases are 1, 2, 3, 4, and 4, with mean $14/5 = 2.8$.

The total number of comparisons for the whole array is $Y = C_1 + C_2 + \ldots + C_{N-1}$. Expectation distributes over a sum: the expected value of a sum is the sum of the expected values. So, without having to deal with any relationship between the passes, we can write

$$E(Y) = \Sigma_{j=1}^{N-1} E(C_j) = \Sigma_{j=1}^{N-1} [(j + 2)/2 - 1/(j + 1)]$$

The first part is $(1/2) \cdot [N(N - 1)/2 + 2(N - 1)] = (N^2 + 3N)/4 - 1$. The second part is $1/2 + 1/3 + \ldots + 1/N = H_N - 1$, where $H_N = 1 + 1/2 + \ldots + 1/N$ is the **harmonic number**. The harmonic number grows slowly: $H_N = \ln N + O(1)$. When the two parts are combined, the −1s cancel:

Theorem 19.1 (Average of insertion sort)

If all $N!$ arrangements of N distinct keys are equally likely, the expected number of comparisons of the `insertion` routine is

$$E(Y) = (N^2 + 3N)/4 - H_N \approx N^2/4$$

The expected number of shifts is $N(N - 1)/4$.

Proof. The number of comparisons is the computation above. For the number of shifts the same route is followed: if the element lands at the kth position from the right, $k - 1$ shifts are made, for $k = 1, \ldots, j + 1$. The mean is $j/2$. Summed over $j = 1, \ldots, N - 1$, the total is $N(N - 1)/4$.

The formula was checked for small values of N by counting all arrangements one by one. In Table 19.5 the "all arrangements" column was found by sorting each of the $N!$ permutations with an instrumented `insertion` and averaging the comparison counts. The result is exactly the same as the formula.

N	$N!$	All arrangements	Formula	$N^2/4$	Average shifts
3	6	2,667	2,667	2,25	1,5
4	24	4,917	4,917	4,00	3,0
5	120	7,717	7,717	6,25	5,0
6	720	11,050	11,050	9,00	7,5
7	5.040	14,907	14,907	12,25	10,5

N	*N!*	All arrangements	Formula	$N^2/4$	Average shifts
8	40.320	19,282	19,282	16,00	14,0

Table 19.5: Average number of comparisons of insertion sort. All *N!* arrangements were counted with a C program.

For small N the formula is clearly larger than $N^2/4$, because the $3N/4$ term still matters. As N grows, $N^2/4$ dominates. For $N = 1{,}000$ the formula gives $250{,}750 - 7.49 \approx 250{,}743$; that is about half of the worst case (499,500). The average case is twice as good as the worst case, but is still $\Theta(n^2)$. Shifts also number about $n^2/4$ on average. A shift is sometimes called a "half swap": in a swap two elements trade places, in a shift only one does.

Note the assumption the average-case analysis rests on: all arrangements are equally likely. Real data may not fit this assumption. If an application's data is mostly nearly sorted, the real average of insertion sort is far smaller than $n^2/4$. The analysis is done on a model; if the model does not fit the data, neither does the result.

19.4 Bubble sort

19.4.1 Idea and code

Bubble sort compares only neighboring elements and swaps them if they are in the wrong order. In a pass that moves from the right end of the array to the left, a small element moves one position left with each neighbor swap. When the pass is over, the smallest of the remaining elements has reached the far left, just like an air bubble rising from the bottom of the water to the surface. The algorithm's name comes from this analogy.

```
void bubble(int a[], int L, int R) {
    int i, j;
    for (i = L; i < R; i++)
        for (j = R; j > i; j--)
            if (a[j] < a[j-1]) SWAP(a[j], a[j-1]);
}
```

- **Line 3.** The outer loop runs $n - 1$ passes. At the end of pass `i`, the smallest of the remaining elements is in `a[i]`.
- **Line 4.** The inner loop brings `j` down from the right end of the array to `i + 1`. `a[L..i-1]` was settled in earlier passes; that region is not entered.
- **Line 5.** Compare-exchange: if the right element `a[j]` is smaller than the left `a[j - 1]`, they trade places. The comparison is strict; equal neighbors are not swapped.

Selection sort also puts the smallest of the remaining elements into `a[i]` in every pass. The difference lies in the method. Selection sort finds the minimum and carries it with a single long-distance swap. Bubble sort carries the small element step by step to the left with neighbor swaps. During this journey other elements also change places. For example, large elements drift a little to the right in every pass. For this reason bubble sort makes many more moves than selection sort, but in return it gains an important property: because only neighbors are swapped, it is stable.

Figure 19.7 shows the passes of the call `bubble(a, 0, 4)` on *a = [7, 2, 9, 5, 1]*; Table 19.6 gives each comparison separately.

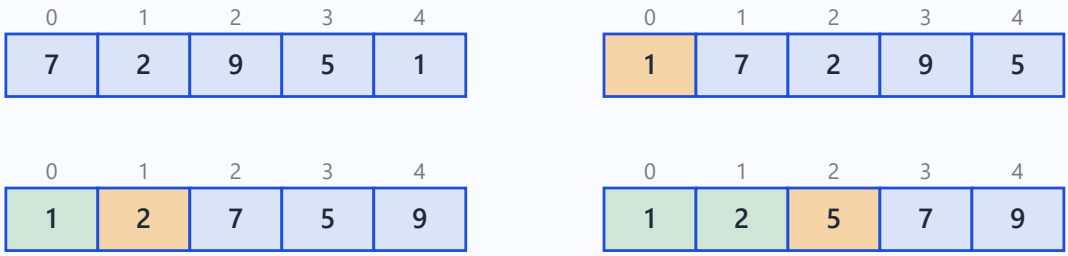


Figure 19.7: The end of each pass of bubble sort, left to right and top to bottom. (1) start. (2) After pass 0, 1 has risen to the far left. (3) After pass 1, 2 is in place. (4) After pass 2 the array is sorted; pass 3 changes nothing. The orange box is the element the pass settled.

Pass `i`	`j`	Test `a[j] < a[j-1]`	Array
0	4	1 < 5 true, swap	[7, 2, 9, 1, 5]
0	3	1 < 9 true, swap	[7, 2, 1, 9, 5]
0	2	1 < 2 true, swap	[7, 1, 2, 9, 5]
0	1	1 < 7 true, swap	[**1**, 7, 2, 9, 5]
1	4	5 < 9 true, swap	[1, 7, 2, 5, 9]
1	3	5 < 2 false	[1, 7, 2, 5, 9]
1	2	2 < 7 true, swap	[1, **2**, 7, 5, 9]
2	4	9 < 5 false	[1, 2, 7, 5, 9]
2	3	5 < 7 true, swap	[1, 2, **5**, 7, 9]
3	4	9 < 7 false	[1, 2, 5, **7**, 9]

Table 19.6: Trace of `bubble(a, 0, 4)`, *a = [7, 2, 9, 5, 1]*.

In pass 0, 1 rose to the far left with four swaps. In pass 1 two things happened at once: 5 and 9 traded places and 9 reached the far right, then 2 rose into its place. In pass 2 a single swap sorted the array. Pass 3 made only one comparison and changed nothing; 7 was already in `a[3]`. In total there were 10 comparisons and 7 swaps. The number of swaps is again 7, just like the number of shifts of insertion sort on the same array.

19.4.2 The early-exit version: `bubble2`

The last pass in the trace was in vain: the array was sorted after pass 2, but the algorithm did not know it. Yet the information was at hand. If no swap was made during a pass, every neighboring pair is in the right order, that is, the array is sorted. There is no need to enter the following passes. The code can be written a little more carefully with this observation:

```
void bubble2(int a[], int L, int R) {
    for (int i = L; i < R; i++) {
        int swapped = 0;      // swap in this pass?
        for (int j = R; j > i; j--)
            if (less(a[j], a[j - 1])) {
                SWAP(a[j], a[j - 1]);
                swapped = 1;
            }
        if (!swapped) return; // sorted: exit early
    }
}
```

- **Line 3.** At the start of each pass the `swapped` flag is cleared.
- **Lines 5–8.** When a swap is made, the flag becomes 1. This time the comparison is written with the `less` macro; its meaning is the same as the line in `bubble`.
- **Line 9.** If the pass ended without a swap, the routine returns immediately.

Table 19.7 traces the call `bubble2(a, 0, 4)` on *a = [2, 1, 3, 5, 4]*. In this array only two pairs are in the wrong order.

Pass `i`	`j`	Test	`swapped`	Array
0	—	start of pass	0	[2, 1, 3, 5, 4]
0	4	4 < 5 true, swap	1	[2, 1, 3, 4, 5]
0	3	4 < 3 false	1	
0	2	3 < 1 false	1	
0	1	1 < 2 true, swap	1	[1, 2, 3, 4, 5]
1	—	start of pass	0	
1	4	5 < 4 false	0	
1	3	4 < 3 false	0	
1	2	3 < 2 false	0	
1	—	`!swapped` true, `return`	0	[1, 2, 3, 4, 5]

Table 19.7: Trace of `bubble2(a, 0, 4)`, *a = [2, 1, 3, 5, 4]*.

The first pass sorted the array with two swaps. The second pass made no swap and the routine returned. In total 4 + 3 = 7 comparisons were made; without early exit, two more passes (2 + 1 = 3 comparisons) would have run in vain. On sorted input, `bubble2` finishes in a single pass with $n - 1$ comparisons; its best case is $\Theta(n)$.

The gain from early exit depends on the shape of the input. In this version, which moves right to left, a small element can go all the way to the left in one pass, but a large element advances only one position right per pass. In the array *[2, 3, 4, 5, 1]*, 1 rises to the front in the first pass and the routine stops after the second: 2 passes. In the array *[5, 1, 2, 3, 4]*, however, four passes are needed for 5 to reach the far right; early exit gains nothing. In both arrays only four pairs are in the wrong order.

19.4.3 Analysis

`bubble` makes $(n - 1) + (n - 2) + \ldots + 1 = n(n - 1)/2$ comparisons in every case; the bounds of the inner loop do not depend on the input. The number of swaps does depend on the input. On sorted input no swaps are made. On reverse-sorted input every comparison results in a swap, and $n(n - 1)/2$ swaps are made. On random input the average number of swaps is $n(n - 1)/4$; the reason is explained right below with inversions.

n	Comparison	Swaps (worst)	Swaps (average)
4	6	6	3
5	10	10	5
6	15	15	7,5
7	21	21	10,5

Table 19.8: Comparison and swap counts of bubble sort (`bubble`). The average column was found by counting all *n!* arrangements.

Some sources, for example Sedgewick's book, say that bubble sort also makes about $n^2/2$ exchanges on average. The count in Table 19.8, however, shows that the average number of swaps is $n(n - 1)/4$, exactly half of the worst case. For the number of comparisons the two statements agree: $n^2/2$.

`bubble2` with early exit is $O(n)$ on sorted input, but it gains little on average. Counted over all arrangements, the average number of passes for $n = 8$ is 5.79; it could have run at most 7 passes. The average number of comparisons is 26.0, very close to the 28 comparisons of the version without early exit. That is why the complexity summary gives the average of bubble sort as $\approx n^2/2$ comparisons. Insertion sort is satisfied with half of that on the same average input.

Bubble sort is stable: only neighbors are swapped, and equal neighbors are not swapped. Two equal elements can never jump over each other, so their relative order does not change. The algorithm is in place; the extra memory is $O(1)$. The total complexity is $O(n^2)$.

In practice bubble sort is rarely used. Insertion sort does the same job with fewer comparisons and cheaper shifts. Bubble sort has two merits: it is easy to teach, and its early-exit version tells you in a single pass whether an array is sorted.

19.5 Inversions: a common measure

Insertion sort sorted *[7, 2, 9, 5, 1]* with 7 shifts, and bubble sort with 7 swaps. That two different algorithms make the same number of moves is no coincidence. Both count the same thing: the pairs in the array that "stand upside down".

Definition 19.4

In an array a, an index pair (i, j) with $i < j$ and $a[i] > a[j]$ is called an **inversion**. The inversion count of the array is the number of such pairs.

Written by value, the inversions of the array *[7, 2, 9, 5, 1]* are (7, 2), (7, 5), (7, 1), (2, 1), (9, 5), (9, 1), and (5, 1). Figure 19.8 shows these seven pairs with arcs. A sorted array has 0 inversions. In a reverse-sorted array every pair is an inversion, and $n(n-1)/2$ pairs can be chosen from n elements. The inversion count is therefore a measure of "how unsorted" an array is: it ranges from 0 to $n(n-1)/2$.

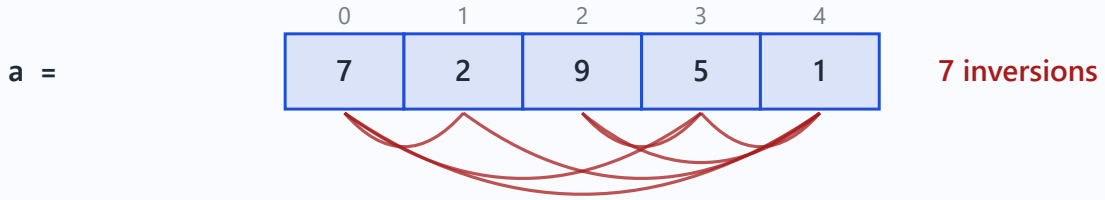


Figure 19.8: The 7 inversions in the array *[7, 2, 9, 5, 1]*. Each arc joins a pair in which the left element is greater than the right one.

Theorem 19.2 (Adjacent swap)

Swapping two adjacent elements that are in the wrong order reduces the inversion count by exactly 1. Every shift in insertion sort also reduces the inversion count by exactly 1. Hence the number of swaps of bubble sort and the number of shifts of insertion sort are equal to the inversion count of the input.

Proof. Let the adjacent pair *a[k] > a[k + 1]* be swapped. The pair these two elements form was an inversion and is not after the swap: a decrease of 1. Now consider any other pair. If neither of its elements is at *k* or *k + 1*, the pair is unaffected. Suppose one of its elements is outside, for example at a position *m < k*. *a[m]* is to the left of both elements, before and after the swap. The number of inversions it forms with these two elements does not change; only which position it is paired with changes. The same holds for *m > k + 1*. The total change is exactly −1.

In insertion sort, when *a[j]*, which is greater than *v*, is shifted one place right, *v* ends up to the left of that element. This removes the inversion *(a[j], v)* and does not change the order of any other pair; the relative order of the other elements is preserved. The inversion count decreases by 1 per shift. Since the algorithms stop when the array is sorted, that is, when the inversion count is 0, the number of moves equals the inversion count.

Applying the theorem to the average case is completed by finding the average inversion count of a random array. For each pair *(i, j)*, one of the two elements is greater; in a random arrangement the probability that the greater one is on the left is *1/2*. To see this, it is enough to pair each arrangement with its reversal: a pair is an inversion in exactly one of the two. Each of the *n(n − 1)/2* pairs contributes *1/2* inversion on average. The average inversion count is

$$n(n-1)/2 \cdot 1/2 = n(n-1)/4$$

The average number of swaps of bubble sort and the average number of shifts of insertion sort are therefore *n(n − 1)/4*. The values 3, 5, 7.5, and 10.5 in the tables come exactly from this formula.

This result has a strong consequence. Any algorithm that only swaps adjacent elements, or shifts elements one at a time, removes at most one inversion per move. The average input has *n(n − 1)/4* inversions. So every such algorithm does $\Omega(n^2)$ work on average; a cleverer loop arrangement cannot get past this wall. The only way past the wall is to remove more than one inversion in a single move. That happens by swapping elements that are far apart. Shell sort and quicksort take this path.

Selection sort is outside this rule; it already makes long-distance swaps. A single long-distance swap can remove many inversions at once. For example, in the array *[5, 2, 3, 4, 1]*, swapping 5 and 1 destroys 7 inversions at once and sorts the array. But the problem of selection sort is not moves but comparisons: regardless of the input it makes $n^2/2$ comparisons.

19.6 Counting sort

19.6.1 Sorting without comparisons

So far the algorithms have learned about the input only by comparing two elements. Counting sort takes a completely different route: it never compares elements with each other. Instead it uses the values as array indices. The algorithm goes back to Harold H. Seward's work in 1954. It has a single condition: the values to be sorted are integers and their range is known, for example *0 … k*.

The idea is this. If, for every value *x*, we know how many elements in the input are less than or equal to *x*, we also know the place of *x* in the output. If there are three elements in the input less than or equal to 3, the last 3 goes to the third position of the output, that is, index 2. To find these counts, first the number of occurrences of each value is counted, then the counts are summed from left to right.

```
int counting_sort(const int a[], int b[],
                  int r, int k) {
    int *c = malloc((k + 1) * sizeof *c);
    if (c == NULL) return -1;
    for (int i = 0; i <= k; i++) c[i] = 0;  // L1
    for (int j = 0; j <= r; j++) c[a[j]]++; // L2
    for (int i = 1; i <= k; i++)            // L3
        c[i] += c[i - 1];
    for (int j = r; j >= 0; j--) {          // L4
        b[c[a[j]] - 1] = a[j];
        c[a[j]]--;
    }
    free(c);
    return 0;
}
```

Name	Meaning
`a[]`	input array; because it is `const`, the routine does not modify it
`b[]`	sorted output array; allocated by the caller, it must have at least *r + 1* elements
`c[]`	auxiliary counter array of *k + 1* elements; allocated inside the routine and released again
`r`	last index: `a[0..r]` is sorted, that is, *r + 1* elements
`k`	the values lie in the range *0 … k*

Table 19.9: Parameters and local array of the `counting_sort` routine.

- **Lines 3–4.** The counter array is allocated from the heap. The reason is that `k` is known only at run time. The `int c[K]` form seen in textbooks needs a size that is constant at compile time. A variable-length array (`int c[k + 1]`) exists in C99, but it takes space on the stack and overflows the stack if `k` is large. If `malloc` fails, the routine returns *–1* and the caller learns of the problem.
- **Line 5 (L1).** The counters are cleared. `malloc` does not clear memory; without this line the counters would start with random values. `calloc` could have done the same job.
- **Line 6 (L2).** Each element increments the counter of its own value. In the expression `c[a[j]]++` the value is used as an index. There is no comparison.
- **Lines 7–8 (L3).** The counters are summed from left to right. When the loop ends, `c[x]` is the number of elements in the input less than or equal to *x*. This kind of accumulation is called a **cumulative sum** (prefix sum).

- **Lines 9–12 (L4).** The input is traversed from right to left. The place of `a[j]` is `c[a[j]] - 1` (1 is subtracted because indices start at 0). After an element is written, its counter is decremented. The next element with the same value lands one place to the left.
- **Lines 13–14.** The counter array is released and 0 is returned.

19.6.2 Counting and cumulative sums

Let the routine be called with *a = [3, 0, 6, 9, 4, 3, 10, 5]*, *r = 7*, *k = 10*. L2 counts each element: `c[3]` becomes 1 first and 2 when the second 3 arrives; the counters of 0, 4, 5, 6, 9, and 10 are 1; 1, 2, 7, and 8 are absent from the input and their counters stay 0. L3 accumulates the counters. Table 19.10 shows the end of the two phases, and Figure 19.9 shows the three arrays in memory.

Index *x*	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
End of L2 (count)	1	0	0	2	1	1	1	0	0	1	1
End of L3 (≤ x)	1	1	1	3	4	5	6	6	6	7	8

Table 19.10: The array `c` at the end of L2 and L3, *a = [3, 0, 6, 9, 4, 3, 10, 5]*, *k = 10*.

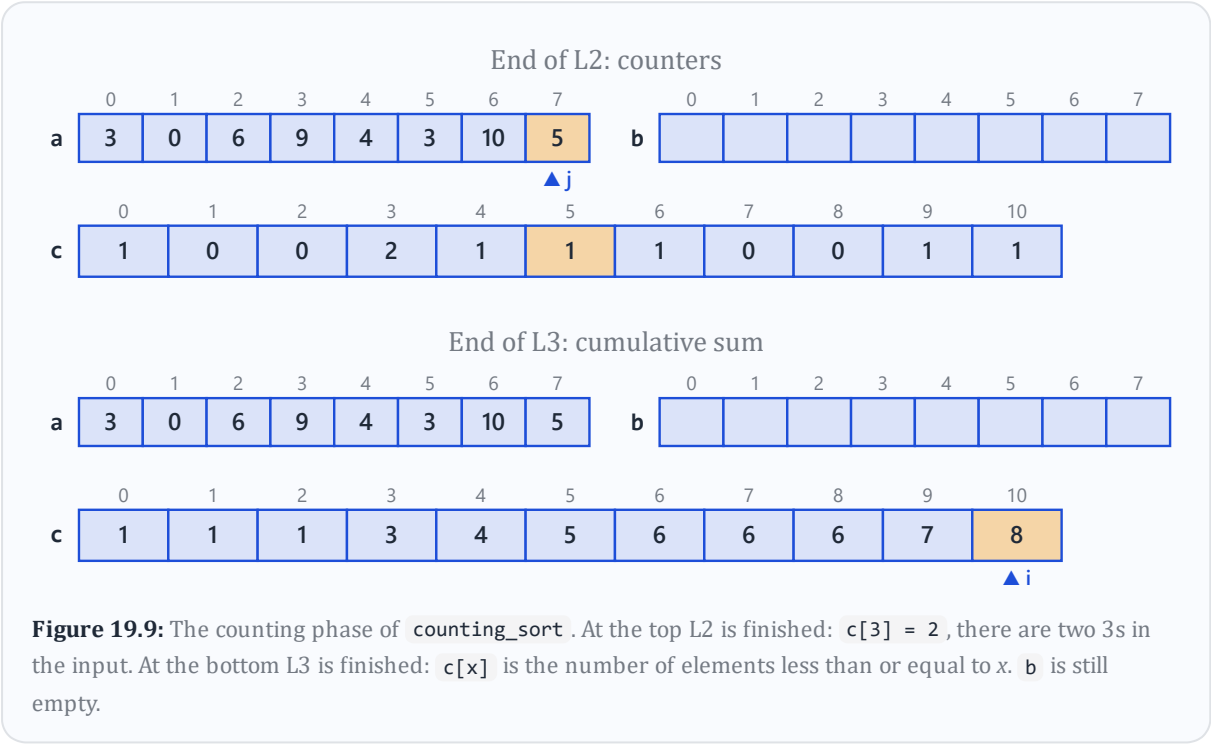


Figure 19.9: The counting phase of `counting_sort`. At the top L2 is finished: `c[3] = 2`, there are two 3s in the input. At the bottom L3 is finished: `c[x]` is the number of elements less than or equal to *x*. `b` is still empty.

The row at the end of L3 can be read directly. `c[9] = 7`: there are 7 elements in the input less than or equal to 9, so 9 goes to the 7th position of the output, that is, `b[6]`. `c[3] = 3`: the last of the two 3s goes to `b[2]`, the first to `b[1]`. The counters of values not in the input (1, 2, 7, 8) repeat the previous one; these values write nothing to the output.

19.6.3 Placement and stability

L4 traverses the input from right to left. Table 19.11 gives each step.

`j`	`a[j]`	`c[a[j]]` before	Written	`c[a[j]]` after
7	5	5	`b[4] = 5`	4
6	10	8	`b[7] = 10`	7
5	3	3	`b[2] = 3`	2
4	4	4	`b[3] = 4`	3
3	9	7	`b[6] = 9`	6
2	6	6	`b[5] = 6`	5
1	0	1	`b[0] = 0`	0
0	3	2	`b[1] = 3`	1

Table 19.11: Trace of L4. `b[c[a[j]] - 1] = a[j]`, followed by `c[a[j]]--`.

The result is *b = [0, 3, 3, 4, 5, 6, 9, 10]*. Figure 19.10 shows the end of L4. When L4 finishes, `c[x]` holds the *first* position of each value in the output: `c[3] = 1` says the first 3 is at `b[1]`.

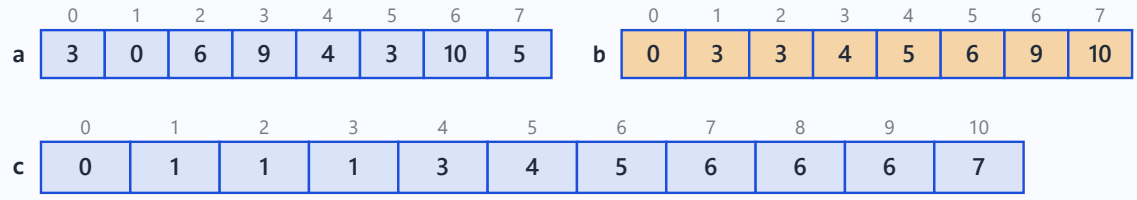


Figure 19.10: At the end of L4, `b` is sorted. The counters, decremented after each write, now show the first position of each value.

Look at the two 3s. The 3 at `a[5]` was processed first and written to `b[2]`. The 3 at `a[0]` was processed later and, since the counter had decreased by one, was written to `b[1]`, one place to the left. The 3 that comes first in the input (`a[0]`) also comes first in the output. The algorithm is stable, and the reason is that L4 traverses from right to left: of the elements with the same value, the rightmost is placed first and takes the rightmost place.

If L4 wrote from left to right, the output would still be sorted, but the order of equal elements would be reversed. In that case the 3 at `a[0]` is processed first and written to `b[2]`, and the 3 at `a[5]` to `b[1]`. When bare numbers are sorted the difference is invisible. When counting sort sorts records, or one pass of radix sort, the difference breaks the result.

Common mistake

Not checking the value range. If the input contains a negative value or a value greater than `k`, `c[a[j]]++` writes outside the array `c`. C does not catch this: the program either silently corrupts another variable or crashes. The routine's contract is "all values lie in the range *0 … k*". If the data may not satisfy this contract, first find the minimum and maximum, shift the values by the minimum, and choose `k` accordingly.

19.6.4 Analysis and limits

The routine has four loops and none of them is nested. Their costs are collected in Table 19.12.

Loop	Work	Cost
L1	clear `c`	*O(k)*
L2	count	*O(n)*
L3	cumulative sum	*O(k)*
L4	place	*O(n)*

Table 19.12: The loops of counting sort; *n = r + 1*.

The total is *O(n + k)*. If *k = O(n)*, that is, if the value range is not much larger than the number of elements, counting sort runs in linear time. The extra memory is *k + 1* elements for `c` and *n* for `b`, in total *O(n + k)*. The algorithm is not in place.

Linear time comes at a price: the data range must be known and small. If the range grows, the *k* term dominates. Suppose 10 numbers are to be sorted, but the values can be anywhere between 0 and 1,000: L1 and L3 are loops of a thousand iterations. For *n = 10*, *k = 1,000 = n^3*; the "linear" algorithm turns into an n^3 algorithm on this input. Insertion sort sorts the same 10 numbers with at most 45 comparisons. For random 32-bit integers, $k \approx 4 \cdot 10^9$ and the counter array would take 16 GB. Counting sort is therefore meant for narrow-range data such as age, grade, day, or letter code. Integers with a wide range are sorted with radix sort: the number is split into digits and each digit is processed by a narrow-range counting sort.

The algorithm's two-pass structure appears elsewhere too: in the first pass frequencies are counted, in the second pass each element is written directly to its computed place. The fast transpose of sparse matrices is exactly this. First the number of nonzero terms in each column is counted. The cumulative sum of the counts gives the starting position of each column in the transposed matrix. In the second pass each term is written directly into place, and the whole job finishes with a single output array in *O(number of columns + number of terms)* time.

19.7 Shell sort

19.7.1 Idea and code

The weakness of insertion sort is that it advances an element only one position per step. A small element at the end of the array needs $n - 1$ shifts to reach the front. In terms of inversions, each shift removes a single inversion, and a reverse-sorted array has $n(n - 1)/2$ inversions. In 1959 Donald Shell proposed a simple remedy: first sort elements that are far apart, then reduce the gap.

Choose a number h. The array is divided into h separate chains made of elements with h positions between them: *a[0], a[h], a[2h], ...* is the first chain, *a[1], a[h+1], ...* the second. Each chain is sorted separately with insertion sort. As a result the array is ***h*-sorted**: no element is greater than the element h places after it. This step starts with a large h, and h is reduced at each step. In the last step $h = 1$; this is ordinary insertion sort.

```
void shell_sort(int a[], int n) {
    for (int h = n / 2; h > 0; h /= 2) // gap
        for (int i = h; i < n; i++) {
            int v = a[i], j = i;
            while (j >= h && a[j - h] > v) {
                a[j] = a[j - h];    // shift back by h
                j -= h;
            }
            a[j] = v;
        }
}
```

- **Line 2.** The gap starts at $n/2$ and is halved at each step: 4, 2, 1 for $n = 8$; 500, 250, 125, 62, 31, 15, 7, 3, 1 for $n = 1{,}000$. This is the gap sequence Shell himself proposed. Thanks to integer division, the sequence always ends with 1.
- **Lines 3–9.** The inner part is insertion sort written with step h instead of 1. `v = a[i]` is set aside; `j` goes back in steps of h; elements greater than `v` are shifted h positions forward. The `j >= h` in the condition ensures that `a[j - h]` is inside the array.

In the code the chains are not sorted separately. The `i` loop goes in order from h to $n - 1$ and inserts each element into place in its own chain. The chains thus advance interleaved, element by element. The result is the same as sorting the chains one by one, because the insertion sort of each chain processes its elements in order from left to right.

The correctness of the algorithm comes from the last step: the $h = 1$ pass is a complete insertion sort and sorts any array. The earlier passes only reduce the work of this last pass. Whichever gap sequence is chosen, the result is correct as long as it ends with 1. The gap sequence determines only the speed.

19.7.2 Step-by-step trace

The call `shell_sort(a, 8)` is traced on *a = [35, 33, 42, 10, 14, 19, 27, 44]*. Table 19.13 gives the array at the end of each gap and the work done.

h	Chains (indices)	Events	Comp.	Shifts	End of gap
4	{0,4} {1,5} {2,6} {3,7}	14, 19, 27 each moved back once by 4; 44 stays	4	3	[14, 19, 27, 10, 35, 33, 42, 44]
2	{0,2,4,6} {1,3,5,7}	10 moved back once by 2; the others stay	6	1	[14, 10, 27, 19, 35, 33, 42, 44]
1	single chain	10, 19, and 33 each moved back once by 1	9	3	[10, 14, 19, 27, 33, 35, 42, 44]
Total			**19**	**7**	

Table 19.13: Trace of `shell_sort(a, 8)`, *a = [35, 33, 42, 10, 14, 19, 27, 44]*. A shift moves an element forward by *h* positions.

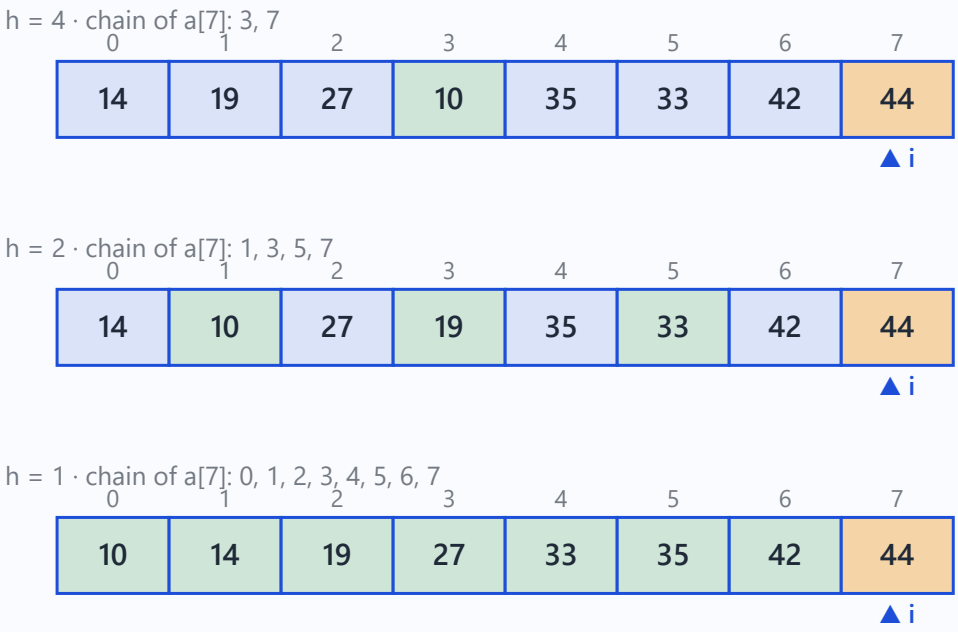


Figure 19.11: The end of three gaps in Shell sort. Top, *h = 4*: each of the four chains is sorted (the last chain processed is 3, 7). Middle, *h = 2*: the two chains are sorted. Bottom, *h = 1*: the array is sorted.

In the *h = 4* pass, `v = 14` is taken for *i = 4*. Since `a[0] = 35 > 14`, 35 shifts to `a[4]` and `j = 0`. `j >= 4` is false; 14 is written to `a[0]`. With a single shift, 14 went four positions left. The same shift also carried 35 four positions right. In insertion sort this would take four neighbor steps. For *i = 5* and *i = 6*, 19 and 27 pass ahead of 33 and 42 in the same way. At *i = 7*, `a[3] = 10 ≤ 44` and 44 stays in place.

In the $h = 2$ pass the array splits into two chains: the even indices (14, 27, 35, 42) are already sorted. Among the odd indices (19, 10, 33, 44) only 10 passes ahead of 19. When the $h = 1$ pass begins, the array is *[14, 10, 27, 19, 35, 33, 42, 44]*. Every element is at most one position from its final place; insertion sort finishes the job with three short shifts.

Plain insertion sort sorts the same array with 18 comparisons and 13 shifts; the array has 13 inversions. Shell sort made 19 comparisons and 7 shifts. In so small an array there is no gain in comparisons, but the number of shifts is halved. 7 shifts removed 13 inversions: long-distance shifts destroy several inversions in a single move. For example, the first shift at $h = 4$ made 35 jump over 33, 42, and 10. As the array grows, this effect becomes pronounced.

19.7.3 Gap sequences and complexity

The running time of Shell sort depends on the gap sequence, and a full analysis is hard. Some of the known results are these:

- For Shell's sequence $n/2, n/4, \ldots, 1$ the worst case is $\Theta(n^2)$. The problem is that when n is a power of 2, all the gaps are even until the last pass. Odd and even positions never meet until 1, and on a bad input the last pass again does work of order n^2.
- For Knuth's sequence $1, 4, 13, 40, 121, \ldots$ ($h \leftarrow 3h + 1$) the worst case is $O(n^{3/2})$. Since consecutive gaps share no common divisor, the chains mix with each other.
- With more complex sequences (for example Pratt's gaps of the form $2^p 3^q$) the worst case drops to $O(n \lg^2 n)$. No closed formula is known for the average case.

One reason Shell sort works is this property: an h-sorted array stays h-sorted when it is k-sorted. That is, later passes do not destroy the order gained by earlier passes, but add to it. The proof of this property is beyond the scope of this note; it is given in Knuth's book.

Table 19.14 compares two gap sequences on $n = 1{,}000$ elements. The values were measured with an instrumented C program.

Gap sequence	Sorted	Reverse sorted	Random
Shell: 500, 250, ..., 1	8.006 / 0	11.716 / 4.700	15.218 / 7.700
Knuth: 364, 121, 40, 13, 4, 1	5.457 / 0	8.550 / 3.920	13.693 / 8.646
Insertion sort (for comparison)	999 / 0	499.500 / 499.500	241.305 / 240.312

Table 19.14: Shell sort for $n = 1{,}000$: comparisons / shifts. The random input was produced by a shuffle; on another random input the numbers vary slightly.

On random input, insertion sort makes more than 240 thousand shifts; both Shell versions stay under 10 thousand. On sorted input, however, insertion sort is unrivaled, because Shell traverses the whole array once for each gap: for $n = 1{,}000$ and Shell's gaps, $500 + 750 + 875 + 938 + 969 + 985 + 993 + 997 + 999 = 8{,}006$ comparisons. That is work of order n per gap, that is, $O(n \lg n)$.

Shell sort is in place: it uses as much memory as insertion sort. It is not stable. Long-distance shifts can make equal elements jump over each other. The smallest example has four elements: *[2a, 5, 2b, 2c]*. In the *h = 2* pass, `v = 2c` is taken for *i = 3*; since `a[1] = 5 > 2`, 5 shifts right and 2c is written to `a[1]`. The array becomes *[2a, 2c, 2b, 5]*. The *h = 1* pass does not move the equal elements. 2c has passed ahead of 2b.

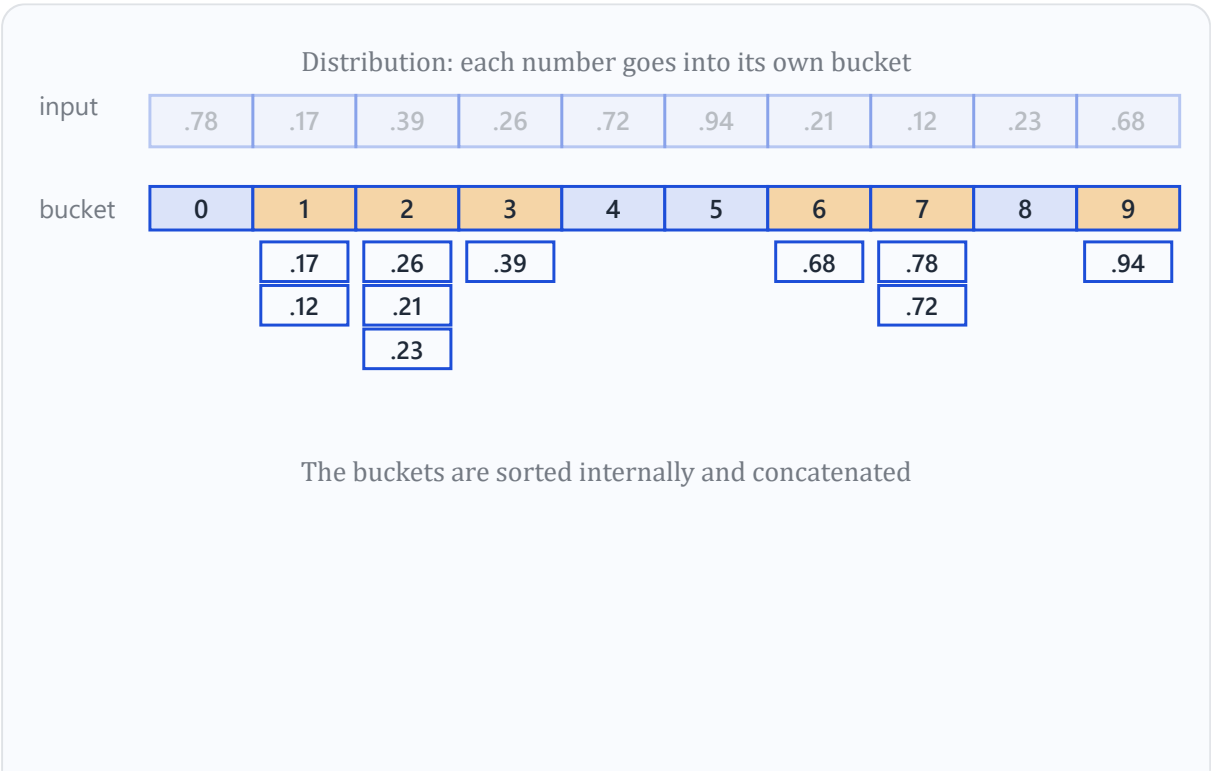
The code is short, needs no extra memory, and on medium-sized arrays is much faster than the n^2 algorithms. These three properties make Shell sort a favorite in embedded systems and small libraries. If the properties of the data are known, Shell sort can also be used together with bucket sort. The elements distributed into buckets are sorted separately in each bucket; a short in-place algorithm is suitable for this job.

19.8 Bucket sort

19.8.1 Idea

Bucket sort carries the idea of counting sort over to real numbers. The input is assumed to be *n* real numbers in the interval *[0, 1)*, distributed uniformly over it. The interval is divided into *n* equal parts, that is, *n* **buckets**. Each number is thrown into its own bucket. Under a uniform distribution, on average one number falls into each bucket. Because the buckets are small, they are cheaply sorted internally with insertion sort. Finally the buckets are concatenated in order.

Figure 19.12 shows this method on an input of 10 numbers. There are 10 buckets and each bucket is 0.1 wide. A value *x* goes to bucket number *⌊10 · x⌋*: 0.78 → 7, 0.17 → 1, 0.39 → 3.



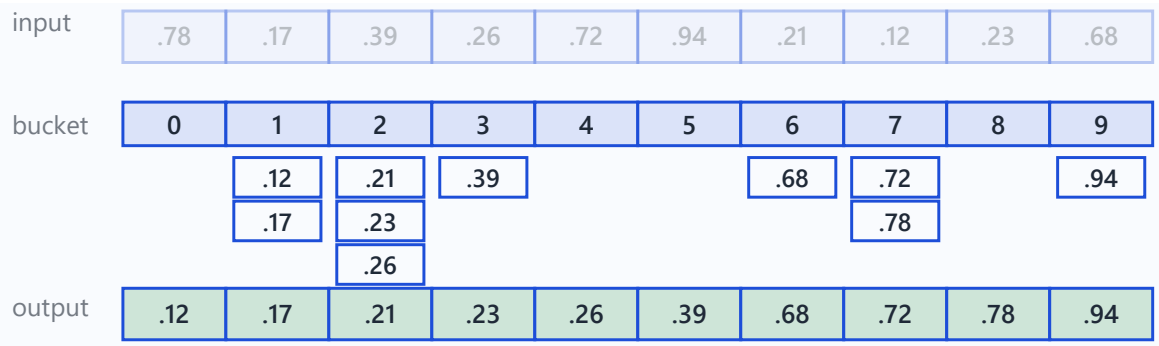


Figure 19.12: Bucket sort. At the top the input has been distributed into ten buckets; the contents of the buckets are still in arrival order (in bucket 1, first .17, then .12). At the bottom the buckets have been sorted internally and the output is formed by reading left to right.

19.8.2 Implementation in C

Each bucket is a linked list. The bucket array holds the head pointers of these lists. The implementation below combines distribution with sorting inside buckets: as each number is thrown into its bucket, it is inserted into its sorted place in the list. This is exactly insertion sort within a bucket.

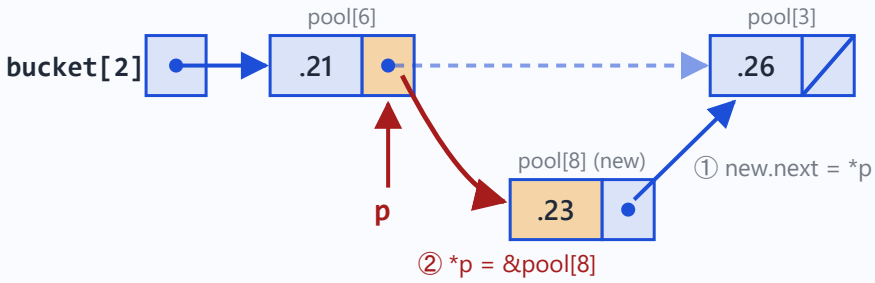
```c
typedef struct bnode { double x; struct bnode *next; } bnode;

int bucket_sort(double a[], int n) {        /* 0 <= a[i] < 1 */
    bnode **bucket = calloc(n, sizeof *bucket);
    bnode *pool = malloc(n * sizeof *pool);
    if (!bucket || !pool) { free(bucket); free(pool); return -1; }
    for (int i = 0; i < n; i++) {           /* distribute + sorted insert */
        bnode **p = &bucket[(int)(a[i] * n)];
        while (*p && (*p)->x <= a[i]) p = &(*p)->next;
        pool[i].x = a[i]; pool[i].next = *p; *p = &pool[i];
    }
    int k = 0;                              /* concatenate buckets */
    for (int b = 0; b < n; b++)
        for (bnode *q = bucket[b]; q; q = q->next) a[k++] = q->x;
    free(bucket); free(pool);
    return 0;
}
```

- **Line 1.** The list node: the value `x` and the link to the next node `next`. The structure is a self-referential `struct`.
- **Line 4.** `bucket` is an array of n pointers of type `bnode *`. `calloc` zeroes the memory; all buckets start as `NULL`, that is, empty lists. If `malloc` had been used, the pointers would hold random addresses and the first read of `(*p)->x` would be undefined behavior.

- **Line 5.** The nodes are taken not one by one with `malloc` but from a single `pool` array. `pool[i]` is the node of `a[i]`. Two allocations are made instead of *n* separate ones, and the memory is returned with a single pair of `free` calls. The allocation cost drops and the risk of leaks decreases.
- **Line 6.** If either of the two allocations failed, both are released. `free(NULL)` is safe; there is no need to test separately which one failed.
- **Line 8.** `p` is a *pointer to a pointer* (`bnode **`). Initially it points to the relevant cell in the bucket array, that is, the head pointer of the list. The bucket number is `(int)(a[i] * n)`. Since $0 \le a[i] < 1$, this number is between 0 and $n - 1$.
- **Line 9.** `p` advances along the list: if the pointer it points to is not null and the value of that node is less than or equal to `a[i]`, `p` starts pointing to the `next` field of that node. When the loop ends, `*p` points to the node the new node will go in front of (or to `NULL`); `p` is the address of the link to be changed.
- **Line 10.** The new node is filled in, its link is directed to the node `*p` points to, and `*p` points to the new node. The insertion is complete.
- **Lines 12–14.** The buckets are traversed in order from 0 to $n - 1$; the values of each list are written back to `a` in order.
- **Line 15.** The two allocations are released. Since the nodes were not allocated separately, they are not released separately.

The pointer to a pointer is the subtlest point of this code. If the list were walked with an ordinary single-star pointer, insertion at the head of the list would be a special case. A node inserted at the head changes the head pointer in the bucket array. A node inserted in the middle changes the `next` field of the previous node. Because `p` holds "the address of the pointer to be changed", the two cases are handled by the same line. Figure 19.13 shows this in the insertion of *0.23*.



while: .21 ≤ .23 → p = &(node .21)->next; .26 ≤ .23 false → stop
result: bucket[2] → .21 → .23 → .26 → NULL (links changed, nodes stayed in place)

Figure 19.13: Insertion of 0.23 (`pool[8]`) into bucket 2. The bucket holds 0.21 and 0.26. `p` is first `&bucket[2]`; since *0.21 ≤ 0.23* it advances to the `next` field of the 0.21 node and stops at *0.26 > 0.23*. The new node's link points to 0.26 (①), then `*p` points to the new node (②).

The state in memory at the end of the routine is seen in Figure 19.14. The nodes sit in the `pool` array in input order: the node of 0.12 is `pool[7]`, because 0.12 is at index 7 of the input. The links string the inside of each bucket into a sorted chain. This is where the strength of the linked list lies: elements do not move in memory, only links change.

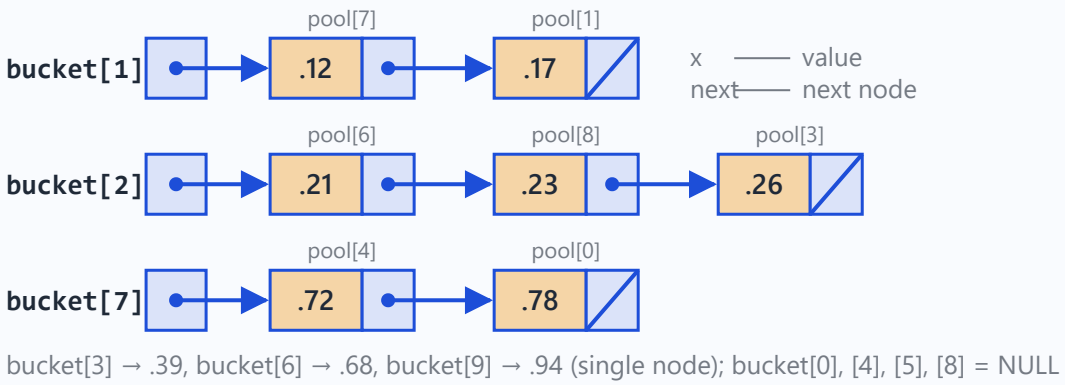


Figure 19.14: Memory before the concatenation loop of `bucket_sort`. Each cell of the bucket array is the head of a list. The labels above the nodes give the node's place in the `pool` array; the diagonal line is a `NULL` link.

Because the comparison is made with `<=`, an equal value is added *after* its equals in the list. The one that comes first in the input stays first; bucket sort is stable in this implementation. If the comparison were `<`, the new element would go in front of its equals and stability would be lost.

19.8.3 Analysis

Distribution and concatenation are $O(n)$: each element is thrown into a bucket once and written back once. The real question is the cost of sorting inside the buckets. If n_i elements fall into bucket i, insertion sort does $O(n_i^2)$ work in that bucket. The total cost is $O(n + \Sigma\, n_i^2)$.

If the input is uniformly distributed, each element falls into each bucket with probability $1/n$. In this case the expected number of elements in a bucket is 1 and the expected value of n_i^2 is $2 - 1/n$ (proved in the book by Cormen and coauthors). Over n buckets the total is $n \cdot (2 - 1/n) = 2n - 1$. The average running time is $O(n)$.

If the uniform-distribution assumption fails, so does the result. In the worst case all elements fall into the same bucket. For example, if 5 numbers are 0.90, 0.91, 0.92, 0.93, and 0.94, then $\lfloor 5x \rfloor$ is 4 for all of them. Five elements are insertion-sorted in a single bucket and the cost rises to $O(n^2)$. The extra memory is $O(n)$ for the bucket array and nodes; the algorithm is not in place.

Bucket sort also uses comparisons, but only inside the buckets. Which bucket an element goes to is found by a computation, $\lfloor n \cdot x \rfloor$. This computation is like comparing the element with all the other elements at once. That is why the algorithm is not caught by the lower bound for comparison sorting.

19.9 A lower bound for comparison sorting

19.9.1 The decision tree

Selection, insertion, bubble, and Shell sort made between n^2 and $n^{3/2}$ comparisons. Is something better possible, and how much better? This question has to be answered not for one algorithm but for *all* algorithms based on comparison. The tool for this is the decision tree.

A comparison-based sorting algorithm learns about the input only from calls to `less`. Each call has two outcomes: yes or no. The algorithm's next step depends on this answer. All possible runs of the algorithm can be drawn as a binary tree. This tree is called a **decision tree**.

- Every internal node is a comparison (an ellipse in the drawing): "$a < b$?".
- The node has two children: the left one is the step following the "yes" answer, the right one the step following the "no" answer.
- Every leaf is a decision (a rectangle in the drawing): in which arrangement the input was, that is, in what order the elements are to be written.

For a given input the algorithm starts at the root, goes left or right according to the answer to each comparison, and stops at a leaf. The length of the path from the root to the leaf is the number of comparisons made on that input. Figure 19.15 shows a decision tree that sorts three elements (a, b, c). For $n = 3$ there are $3! = 6$ arrangements and hence 6 leaves.

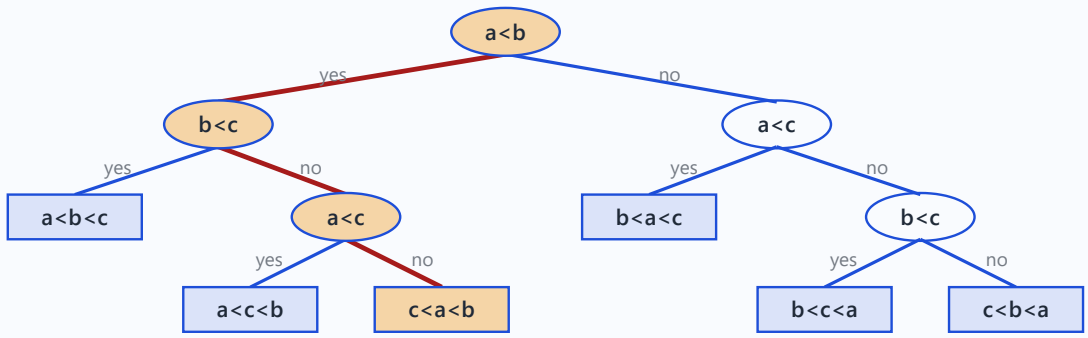


Figure 19.15: Decision tree for three elements. Ellipses are comparisons, rectangles are decisions. The bold path is the one followed by the input $a = 5, b = 8, c = 2$: after 3 comparisons the decision $c < a < b$ is reached.

Let the input be $a = 5, b = 8, c = 2$. The question at the root is "$a < b$?": $5 < 8$, the answer is yes. At the left child "$b < c$?" is asked: $8 < 2$, the answer is no, so c must come before b. At this point $a < b$ and $c < b$ are known, but the order of a and c is not. The third question is "$a < c$?": $5 < 2$, the answer is no. The leaf is $c < a < b$ and the output is 2, 5, 8. For this input 3 comparisons were made.

Some paths of the tree are shorter. If $a < b$ and $b < c$, then $a < c$ is also known by transitivity; the algorithm decides $a < b < c$ with two comparisons. The worst-case number of comparisons is the length of the longest path, that is, the **height** of the tree. The height of this tree is 3.

Every comparison-based sorting algorithm corresponds to such a tree for each n. The tree of insertion sort has one shape, that of selection sort another; but every tree must satisfy the same condition. This condition is the source of the lower bound.

19.9.2 The lower-bound theorem

> **Theorem 19.3 (Lower bound for comparison sorting)**
>
> Every decision tree that sorts n distinct elements has depth (number of levels) at least $\lceil \lg(n!) \rceil + 1$. That is, every comparison-based sorting algorithm makes at least $\lceil \lg(n!) \rceil$ comparisons in the worst case. This number is $\Theta(n \lg n)$.

Proof. **Number of leaves.** Each of the $n!$ arrangements of the input needs a different output: if two different arrangements reached the same leaf, the algorithm would sort one of them wrongly. So the tree has at least $n!$ leaves.

Relation between height and leaves. A binary tree of height h can have at most 2^h leaves. The root is a single node; at each level the number of nodes at most doubles, and after h comparisons at most 2^h different places are reached.

Conclusion. Combining the two observations gives $2^h \geq n!$, that is, $h \geq \lg(n!)$. Since h is an integer, $h \geq \lceil \lg(n!) \rceil$. The number of levels is one more than the height: $\lceil \lg(n!) \rceil + 1$.

Growth rate. Each of the last $n/2$ factors of $n!$ is at least $n/2$; therefore $n! \geq (n/2)^{n/2}$ and $\lg(n!) \geq (n/2) \cdot \lg(n/2)$, that is, $\Omega(n \lg n)$. Also, since $n! \leq n^n$, $\lg(n!) \leq n \lg n$. Together they give $\lg(n!) = \Theta(n \lg n)$.

A sharper approximation comes from Stirling's formula:

$$\lg(n!) = n \lg n - n \lg e + \Theta(\lg n) \approx n \lg n - 1.4427\ n$$

Here $\lg e = 1 / \ln 2 \approx 1.4427$. Table 19.15 puts the lower bound, the Stirling approximation, and the worst case of the basic algorithms side by side. The values were computed in a C program with the sum $\lg(n!) = \lg 2 + \lg 3 + \ldots + \lg n$.

n	$\lceil \lg n! \rceil$	$n \lg n - 1.44n$	$n(n-1)/2$
3	3	0,4	3
4	5	2,2	6
5	7	4,4	10
10	22	18,8	45

n	*⌈lg n!⌉*	*n lg n − 1.44n*	*n(n − 1)/2*
20	62	57,6	190
100	525	520,1	4.950

Table 19.15: The lower bound for comparison sorting and the worst case of the n^2 algorithms.

For small *n* the Stirling approximation falls well below the lower bound, because the discarded *Θ(lg n)* term still matters. At *n = 100* the difference drops to 5. The gulf between the last two columns of the table is also striking: for 100 elements the lower bound is 525, while selection sort makes 4,950 comparisons. The gap shows there is room for better algorithms. Merge sort and heap sort close this gap: in the worst case they make *O(n lg n)* comparisons and are within a constant factor of the lower bound.

From here on, the theorem gives a yardstick, a touchstone, for comparing all sorting algorithms. If a comparison-based algorithm has reached *O(n lg n)*, nothing is better in order of magnitude; only constant factors and practical properties (memory, stability, cache friendliness) can be improved. The same bound holds for the average case: in a decision tree the average depth of the leaves is also at least *lg(n!)*.

The theorem has one assumption: the algorithm learns about the input only through comparisons. Counting sort, bucket sort, and radix sort violate this assumption. The line `c[a[j]]++` does not compare `a[j]` with another element; it uses the value itself as an index and with this single operation makes a *k + 1*-way decision. The decision tree stops being binary and the lower bound loses its validity. The price of linear time comes from here too: for the value to be usable as an index, the range must be known.

19.10 Comparing the algorithms

19.10.1 Counting on the same input

Formulas give the growth rate of the algorithms, but hide constant factors and the shape of the input. The work done by the four comparison-based algorithms on the same 1,000-element inputs sits side by side in Table 19.16. Each cell shows first the comparisons, then the moves. Moves are swaps in selection sort, shifts in insertion sort and Shell sort, and swaps in bubble sort.

Algorithm	Sorted	Reverse sorted	Random
Selection	499.500 / 999	499.500 / 999	499.500 / 999
Insertion	999 / 0	499.500 / 499.500	247.876 / 246.884
Bubble (early exit)	999 / 0	499.500 / 499.500	499.035 / 246.884

Algorithm	Sorted	Reverse sorted	Random
Shell (*n/2, n/4, ...*)	8.006 / 0	11.716 / 4.700	14.760 / 7.254

Table 19.16: Comparison / move counts for $n = 1{,}000$. The random column was counted with a Python simulation (seed 19).

The sorted and reverse-sorted columns follow directly from the formulas and were also found to be exactly the same with this note's C program. The random column itself depends on the shuffle chosen. This note's C program, with a different shuffle, measured these values: insertion 241,305 / 240,312, bubble with early exit 499,247 / 240,312 (977 passes), Shell 15,218 / 7,700. The numbers are slightly different, but the relationships are the same. In both counts the number of shifts of insertion sort equals the number of swaps of bubble sort, because both are the inversion count of the input. The value $n(n-1)/4 = 249{,}750$ predicted by the theorem is close to both measurements.

- **Selection** is blind to the input: in every case it makes 499,500 comparisons and 999 swaps.
- **Insertion** finishes with 999 comparisons on sorted input; it makes $\approx n^2/4$ comparisons on random input and $n^2/2$ on reversed input.
- **Bubble**, even with early exit, makes almost as many comparisons as the worst case on random input: for 1,000 elements only the last 20–30 passes are skipped.
- **Shell**, on random input, reduces comparisons about sixteenfold and shifts by more than thirtyfold relative to insertion sort. Long-distance shifts destroy inversions quickly.

The counts can also be backed by timing measurements. Table 19.17 gives the time, in milliseconds, to sort the same random inputs. Times vary with the machine, the compiler, and the build options; what matters here is not the absolute values but how the time changes when n doubles.

n	Selection	Insertion	Bubble (early exit)	Shell
5.000	39	23	48	1
10.000	159	106	174	2
20.000	719	474	1.018	4
40.000	2.941	1.578	4.712	13

Table 19.17: Time (ms) on random input. Measurement: zig cc `-O2`, Windows, single run. Values vary with the machine.

Each time n doubles, the time of the three basic algorithms grows about fourfold; this is the sign of n^2 growth. Shell sort sorts 40,000 elements in 13 milliseconds. Insertion sort is about twice as fast as selection sort, because its comparison count is half. Bubble sort is the slowest: it makes as many comparisons as selection sort, on top of that as many moves as insertion sort, and each of its moves is a swap of three assignments.

19.10.2 Complexity summary

Algorithm	Best	Average	Worst	Extra memory	Stable	In place
Selection	$n^2/2$	$n^2/2$	$n^2/2$	$O(1)$	no	yes
Insertion	$O(n)$	$\approx n^2/4$	$n^2/2$	$O(1)$	yes	yes
Bubble (early exit)	$O(n)$	$\approx n^2/2$	$n^2/2$	$O(1)$	yes	yes
Shell (Knuth gaps)	$O(n \lg n)$	—	$O(n^{3/2})$	$O(1)$	no	yes
Counting	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(n + k)$	yes	no
Bucket (uniform distribution)	$O(n)$	$O(n)$	$O(n^2)$	$O(n)$	yes*	no

Table 19.18: Summary of this note's algorithms. The numbers in the selection, insertion, and bubble rows are comparison counts.

The asterisk in the table means that the stability of bucket sort depends on the sort used inside the buckets; the implementation here is stable because it inserts with `<=`. The average of Shell sort depends on the gap sequence and is not known by a closed formula. For every comparison-based algorithm the lower bound is $\lceil \lg n! \rceil \approx n \lg n$; none of the first four rows in the table reaches this bound in the worst case.

19.10.3 Which one when?

There is no such thing as "the best sorting algorithm"; each algorithm suits a situation. The choice depends on the size of the data, its shape, and which operation is expensive.

- **Selection sort if swaps or writes are expensive.** It makes exactly $n - 1$ swaps. It suits large records and settings such as flash memory where writing is expensive.
- **Insertion sort if the data is small or nearly sorted.** It is stable, in place, and drops to $O(n)$ on nearly sorted data. Practical versions of quicksort and merge sort leave small parts to insertion sort.
- **Bubble sort for teaching and for testing sortedness.** The early-exit version detects in a single pass that an array is sorted. It rarely appears in production code.
- **Counting sort if the values lie in a small integer range.** It is $O(n + k)$ and stable. If k is much larger than n, memory and time are wasted.
- **Shell sort for medium-sized data when there is no extra memory.** With a few lines of code it is clearly faster than the n^2 algorithms.
- **Bucket sort if the data is uniformly distributed in $[0, 1)$.** It is $O(n)$ on average, but slows down to n^2 if the distribution is skewed.

19.11 Common mistakes

1. **Keeping the value of the minimum in selection sort.** If only the value is kept, as in `int min = a[i];`, the position of the smallest element is unknown at the end of the pass and the swap cannot be made. The swap needs the index (`min = j`).
2. **Not saving `v = a[i]` before shifting in insertion sort.** The first shift overwrites `a[i]` and the value to be inserted is lost.
3. **Writing the bound as `j >= 0`.** If the subarray `a[L..R]` is being sorted, the correct bound is `j >= L`. `j >= 0` corrupts elements outside the part when $L > 0$.
4. **Mixing up the direction in the `if (less(A, B)) SWAP(A, B);` pattern.** This line swaps if $A < B$. If the smaller element is to move left, `A` must be the element on the right: `less(a[j], a[j - 1])`.
5. **Forgetting the range in counting sort.** A negative value or one greater than `k` writes outside `c[]`; C does not catch it.
6. **Writing L4 from left to right in counting sort.** The array is still sorted, but the order of equal elements is reversed and stability is lost.
7. **Writing a multi-statement macro with braces alone.** It gives a compile error in an `if (…) SWAP(…); else …` structure; the body must be wrapped in `do { … } while (0)`.
8. **Taking the average case to be the mean of the best and worst cases.** In bubble sort with early exit, the extremes are $n - 1$ and $n(n - 1)/2$ comparisons; their mean is $\approx n^2/4$. The true average is $\approx n^2/2$: for $n = 8$ the mean of the extremes is 17.5, the average over all arrangements is 26.0. The average is the expected value taken over all arrangements.

19.12 Exercises and solutions

19.12.1 Passes of selection sort

Question. For *[5, 1, 4, 2, 8]*, write the array at the end of each pass of selection sort. How many swaps were made?

Solution. In pass 0 the minimum is 1 (`min = 1`); it is swapped with 5: *[1, 5, 4, 2, 8]*. In pass 1 the smallest of the rest is 2 (`min = 3`); it is swapped with 5: *[1, 2, 4, 5, 8]*. In pass 2 the minimum 4 is already at `a[2]`; the swap is with itself. In pass 3, 5 and 8 are compared; 5 is in place. Over the four passes the array goes *[1, 5, 4, 2, 8]* → *[1, 2, 4, 5, 8]* → *[1, 2, 4, 5, 8]* → *[1, 2, 4, 5, 8]*. `SWAP` is called four times, but only the first two are real swaps. The number of comparisons is 10 as always.

19.12.2 The two extremes of insertion sort

Question. How many comparisons does insertion sort make for *[1, 2, 3, …, n]* and *[n, …, 2, 1]*?

Solution. On sorted input each pass ends with a single comparison: $n - 1$. On reverse-sorted input pass i makes i comparisons: $1 + 2 + \ldots + (n - 1) = n(n - 1)/2$. For $n = 1{,}000$: 999 and 499,500.

19.12.3 Bubble sort with early exit

Question. How many passes does `bubble2` run on the input *[1, 2, 3, 5, 4]*?

Solution. 2 passes. In the first pass, at $j = 4$, $4 < 5$ is true and 5 and 4 are swapped; no swap occurs in the rest of the pass. The array is sorted, but the algorithm learns this only in the second pass: in the second pass no swap occurs and the routine returns. In total $4 + 3 = 7$ comparisons and 1 swap are made.

19.12.4 The counter array

Question. For $a = [2, 5, 3, 0, 2, 3, 0, 3]$ and $k = 5$, write the array `c` of counting sort after L3.

Solution. At the end of L2 the counters are $c = [2, 0, 2, 3, 0, 1]$: two 0s, no 1, two 2s, three 3s, no 4, one 5. L3 accumulates them: $c = [2, 2, 4, 7, 7, 8]$. Reading example: `c[3] = 7`, there are seven elements less than or equal to 3; the last 3 is written to `b[6]`. The sorted output is *[0, 0, 2, 2, 3, 3, 3, 5]*.

19.12.5 Height of the decision tree

Question. What is the minimum height of a decision tree for 3 elements? For 4 elements?

Solution. $\lceil \lg 3! \rceil = \lceil \lg 6 \rceil = 3$ and $\lceil \lg 4! \rceil = \lceil \lg 24 \rceil = 5$. For 3 elements the tree in Figure 19.15 reaches this bound. For 4 elements, 4 comparisons are not enough: 4 comparisons can separate at most $2^4 = 16$ different outcomes, yet there are 24 arrangements. There are algorithms that sort 4 elements with 5 comparisons.

19.12.6 Instability of selection sort

Question. Find the smallest example showing that selection sort is unstable.

Solution. Two elements are not enough: in a two-element array equal keys are not swapped, and for different keys the question of stability does not arise. The three-element *[2a, 2b, 1]* suffices. In the first pass the minimum is 1 and it is swapped with 2a at `a[0]`: *[1, 2b, 2a]*. In the second pass `less(2a, 2b)` is false and the swap is with itself. The result is *[1, 2b, 2a]*: 2b has passed ahead of 2a. The long-distance swap made 2a jump over 2b.

19.13 Summary

- Sorting arranges records by their keys. In internal sorting the data is in memory; in external sorting the cost is determined by disk accesses.

- A stable algorithm preserves the relative order of records with equal keys; sorting on several fields and radix sort rely on this. An in-place algorithm uses $O(1)$ extra memory.
- The `less` and `SWAP` macros let the algorithms be written independently of type and criterion; a multi-statement macro is written inside `do { … } while (0)`.
- Selection sort is blind to the input: in every case $n(n-1)/2$ comparisons and $n-1$ swaps. It is unstable because of the long-distance swap.
- Insertion sort makes $n-1$ comparisons on sorted input and $n(n-1)/2$ on reverse-sorted input. Its average over all $n!$ arrangements is $(n^2+3n)/4 - H_n \approx n^2/4$; it is not the mean of the best and worst cases. It is stable and in place. The sentinel version removes the bounds test in the inner loop.
- Bubble sort swaps neighbors; it is stable. The early-exit version is $O(n)$ on sorted input, but on average still makes $\approx n^2/2$ comparisons.
- An inversion is a pair that stands in the wrong order. An adjacent swap or a one-position shift removes exactly one inversion; since a random array has $n(n-1)/4$ inversions on average, every such algorithm is $\Omega(n^2)$ on average.
- Counting sort makes no comparisons; it runs in $O(n+k)$ time with $O(n+k)$ extra memory. It is stable because placement is done from right to left. It is useless if the range is large.
- Shell sort performs insertion sort with decreasing gaps; the last pass, $h=1$, guarantees correctness. Long-distance shifts remove inversions quickly. Speed depends on the gap sequence; it is in place and unstable.
- Bucket sort is $O(n)$ on average on data uniformly distributed in $[0, 1)$; it rises to $O(n^2)$ if all elements fall into one bucket. The C implementation does sorted insertion with a pointer to a pointer and takes the nodes from a single pool.
- Every comparison sort is a decision tree. Since the tree has at least $n!$ leaves, its height is at least $\lceil \lg n! \rceil = \Theta(n \lg n)$. This bound is the yardstick for advanced sorting algorithms; algorithms that make no comparisons are not caught by it.

19.14 Key terms

Key terms introduced or used in this chapter.

sorting record key stable sort in-place sort macro selection sort insertion sort invariant sentinel permutation bubble sort inversion counting sort radix sort Shell sort bucket bucket sort decision tree lower bound memory leak

19.15 Try it yourself

1. How many comparisons and how many real swaps does the `selection` routine make on the array

[3, 1, 2]? What is the array at the end of each pass?

Answer

3 comparisons and 2 swaps. In pass 0 the minimum is 1: *[1, 3, 2]*. In pass 1, *2 < 3* and a swap is made: *[1, 2, 3]*. Both swaps are real.

2. What is the inversion count of the array *[4, 1, 3, 2]*? With how many shifts and how many comparisons does `insertion` sort this array?

Answer

4 inversions: (4, 1), (4, 3), (4, 2), (3, 2). The number of shifts equals the inversion count: 4. There are 6 comparisons: 1 at *i = 1* (1 goes to the front), 2 at *i = 2*, 3 at *i = 3*.

3. How many passes does `bubble2` run on the arrays *[5, 1, 2, 3, 4]* and *[2, 3, 4, 5, 1]*? Why is there a difference although both arrays have the same inversion count?

Answer

4 passes on the first, 2 on the second. Because the inner loop moves from right to left, a small element can go all the way left in one pass, but a large element advances only one position right per pass. In *[5, 1, 2, 3, 4]*, 5 needs four passes to reach the end.

4. `counting_sort` is called with *a = [1, 0, 1, 2]* and *k = 2*. What is `c` at the end of L3? To which index is the 1 at `a[0]` written in the output?

Answer

c = [1, 3, 4]. L4 traverses from right to left: `a[3] = 2 → b[3]`, `a[2] = 1 → b[2]`, `a[1] = 0 → b[0]`, `a[0] = 1 → b[1]`. The 1 that comes first in the input is also first in the output; the algorithm is stable.

5. What is the expected number of comparisons when the fifth element (*j = 4*) is inserted in insertion sort?

Answer

$E(C_4) = (4 + 2)/2 - 1/5 = 2.8$. The comparison counts of the five cases are 1, 2, 3, 4, and 4; their mean is *14/5 = 2.8*.

6. What is the minimum number of comparisons a comparison-based algorithm sorting 5 elements

must make in the worst case? How many does selection sort make?

Answer

⌈lg 5!⌉ = ⌈lg 120⌉ = ⌈6.91⌉ = 7. Selection sort makes 10 comparisons on every input; three more than the lower bound.

7. `bucket_sort` is called with $n = 5$ and the input 0.94, 0.93, 0.92, 0.91, 0.90. How is it distributed into the buckets, and why is this input bad?

Answer

$\lfloor 5x \rfloor$ is 4 for all five numbers; all of them fall into bucket 4. Since the numbers arrive in decreasing order, each new number goes to the front of the list, so the `while` loop makes only one comparison each time. If the input had arrived in increasing order, each insertion would have walked the whole list. For n elements falling into a single bucket the worst case is $O(n^2)$; the uniform-distribution assumption is violated.

8. Which gaps does `shell_sort` use on a 10-element array? Why is the last gap always 1, and why does it matter?

Answer

5, 2, 1. The gap is halved with integer division at each step and cannot become 0 without passing through 1: when $h = 1$ the next value is 0 and the loop ends. The $h = 1$ pass is a complete insertion sort; whatever the earlier passes did, it sorts the array. The correctness of the algorithm comes from this last pass.

19.16 Further reading

- E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2nd ed., Chapter 7. The sorting concepts, the analysis of insertion sort, and the decision tree follow the layout of this source.
- R. Sedgewick, *Algorithms in C, Parts 1–4*, 3rd ed., Chapter 6. The basic sorting algorithms written with the `less` and `SWAP` macros, insertion sort with a sentinel, and the gap sequences of Shell sort are discussed with detailed experiments.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, Chapters 2 and 8. The loop-invariant correctness proof of insertion sort, counting sort and bucket sort, the average-case analysis of bucket sort, and the lower-bound theorem.
- D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 2nd ed., Section 5.2. An in-depth study of inversions, Shell sort, and h-sorted arrays; the reference work on these topics.

19.17 References

1. E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2nd ed., Silicon Press, 2008. Chapter 7, Sorting.
2. R. Sedgewick, *Algorithms in C, Parts 1–4: Fundamentals, Data Structures, Sorting, Searching*, 3rd ed., Addison-Wesley, 1998. Chapter 6, Elementary Sorting Methods.
3. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022. Chapter 2 (Getting Started) and Chapter 8 (Sorting in Linear Time).
4. D. L. Shell, “A High-Speed Sorting Procedure”, *Communications of the ACM*, 2(7), 30–32, 1959.
5. D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 2nd ed., Addison-Wesley, 1998.

PART V · CHAPTER 20

Advanced Sorting Algorithms

Once the number of items climbs past tens of thousands, the elementary sorts become unusable because their running time grows like n^2. This note studies four algorithms that do the same job in time proportional to $n \lg n$, with C code and step-by-step traces: quicksort, merge sort, heap sort, and radix sort. It also covers the C library routine `qsort` together with its comparison function. The time and memory cost of each algorithm is derived, and the figures quoted here were checked by running the programs.

LEARNING OUTCOMES

Quicksort
State the partition invariant, and trace `partition()` and `quicksort1()` step by step on an array.

Analysis
Derive the best-case $n \lg n$, worst-case n^2, and average-case $\approx 1.39\, n \lg n$ comparison counts from recurrence relations.

Improvements
Keep the stack of the non-recursive version at $O(\lg n)$; show the effect of median-of-three pivoting and a small-subarray cutoff; use `qsort` with a correct comparison function.

Merge sort
Trace `merge`, the frame tree, and the linked (index-based) version; allocate and release the $O(n)$ extra memory with `malloc` and `free`.

Heap sort
Build a heap bottom-up in $O(n)$ and sort in place in $O(n \lg n)$.

Radix sort
Implement least-significant-digit radix sort with linked-list buckets and interpret its complexity.

20.1 Speeding up sorting: divide and conquer

20.1.1 The limit of the elementary sorts

Selection sort finds the smallest remaining item in each pass and puts it in place. Bubble sort compares neighbors and carries the largest item to the end. Each pass fixes exactly one item in its final position. Yet the next pass starts again with a problem of $n - 1$ items. The problem shrinks by only one item per pass, so the total work is $n + (n - 1) + \ldots + 1$, which is of order n^2. Insertion sort runs in linear time on nearly sorted data, but on general input it too reaches n^2.

To break through this barrier it is not enough to place items; the problem itself has to shrink quickly at every step. There is also a known lower bound for every comparison-based sorting algorithm. To identify the correct one of the $n!$ orderings of n items, at least $\lceil \lg n! \rceil \approx n \lg n - 1.44\, n$ comparisons are needed in the worst case. Here $\lg$ is the base-2 logarithm. The algorithms in this note approach that bound to within a small constant factor.

20.1.2 Divide and conquer

> **Definition 20.1**
>
> **Divide and conquer** is a design method that recursively splits a problem into smaller subproblems until they are small enough to solve directly, solves the subproblems, and combines their solutions. It has three steps: *divide*, *conquer*, and *combine*.

Quicksort and merge sort are two different readings of this method. Quicksort does the heavy work while dividing: it splits the array into two parts so that everything on the left is less than or equal to everything on the right. No combining is then needed; once the two parts are sorted separately, the whole array is sorted. Merge sort does the opposite. It splits the array blindly in the middle and does the heavy work when merging two sorted halves.

The cost of both algorithms is written as a **recurrence relation**. The cost of a problem of n items is expressed in terms of the costs of smaller problems and the cost of the dividing or combining work. For example, an algorithm that halves the problem at each step and spends n on the split satisfies $C_n = 2C_{n/2} + n$.

The arrays sorted in this note are always `int` arrays. For comparison and swapping, the two macros used in the elementary sorts are kept unchanged:

```
#define less(A, B) ((A) < (B))
#define SWAP(A, B) do { int t_ = (A); (A) = (B); (B) = t_; } while (0)
```

`less(A, B)` is true when $A < B$. Collecting the comparison in one place makes it easy to add a counter or to change the item type. In this note's measurements, comparisons were counted by adding a counter to this macro. `SWAP` takes two arguments and declares its temporary variable internally. Wrapping the body in `do { … } while (0)` makes the macro behave like a single statement in a place such as `if (…) SWAP(x, y); else …`. If the body were written with braces alone, the trailing semicolon would orphan the `else` and cause a compile error.

20.2 Quicksort

Quicksort is the algorithm published by C. A. R. (Tony) Hoare in 1962. On average it runs in time proportional to $n \lg n$; in the worst case it degrades to n^2. It uses only a small stack as auxiliary storage. It is not **stable**, meaning it does not preserve the relative input order of items with equal keys. Despite this flaw it is one of the fastest known general-purpose sorting algorithms. The name of `qsort` in the standard C library comes from this algorithm.

The code below is taken from R. Sedgewick's *Algorithms in C*. Sedgewick was a doctoral student of Donald Knuth and did extensive work on quicksort. The code is short and carefully written; every line has a job.

20.2.1 `quicksort1`

```
int partition(int a[], int L, int R);

void quicksort1(int a[], int L, int R) {
    if (R <= L) return;
    int i = partition(a, L, R);
    quicksort1(a, L, i - 1);
    quicksort1(a, i + 1, R);
}
```

The routine sorts the range `a[L..R]`. To sort a whole array, call `quicksort1(a, 0, n - 1)`. The prototype on the first line lets the compiler check the call even though the definition of `partition` appears further down.

- `if (R <= L) return;` is the base case of the recursion. If $R < L$ the range is empty, and if $R = L$ it holds a single item. In both cases there is nothing to do.
- `int i = partition(a, L, R);` partitions the range. `partition` puts one item, the **pivot**, at the position it will occupy in the sorted array and returns that position. On return, every item in `a[L..i-1]` is less than or equal to the pivot, and every item in `a[i+1..R]` is greater than or equal to it.
- The two recursive calls sort the left and right of the pivot separately. The pivot is already in place; no later call touches it.

The correctness of the routine follows easily from these three lines. Once the left part is sorted, all of its items stand in the correct order to the left of the pivot, because none of them is greater than the pivot. The same holds for the right part. All the real work is in `partition`.

20.2.2 Partitioning: `partition()`

> **Definition 20.2**
>
> **Partitioning** the range `a[L..R]` means rearranging the items so that, for some position *i*, *a[L..i−1] ≤ a[i] ≤ a[i+1..R]*. The item at *a[i]* is called the pivot. The statement that stays true at every moment while partitioning proceeds is called the **partition invariant**.

```
int partition(int a[], int L, int R) {
    int i = L - 1, j = R, v = a[R]; /* pivot is a[R] */
    for (;;) {
        while (less(a[++i], v))
            ;
        while (less(v, a[--j]))
            if (j == L) break;
        if (i >= j) break;
        SWAP(a[i], a[j]);
    }
    SWAP(a[i], a[R]);
    return i;
}
```

`partition` chooses the rightmost item, `v = a[R]`, as the pivot. It uses two indices: `i` scans from left to right and `j` from right to left. Line by line:

- **Line 2.** The scan starts with `i = L - 1` and `j = R`. Because the scans advance with `++i` and `--j` before looking, the first items examined are `a[L]` and `a[R-1]`. The pivot itself does not take part in the scan.
- **Lines 4–5.** `i` stops at the first item greater than or equal to *v*. This loop has no bounds check, and none is needed. In the worst case `i` reaches `R`, and `a[R] = v` stops the scan because `less(v, v)` is false. Here the pivot acts as a natural **sentinel**.
- **Lines 6–7.** `j` stops at the first item less than or equal to *v*. This scan has no sentinel on its left: if the pivot is the smallest item in the range, `j` could run off the start of the array. The line `if (j == L) break;` prevents that.
- **Line 8.** If `i >= j`, the indices have met or crossed. Every item has been scanned and the loop ends.
- **Line 9.** Otherwise `a[i]` belongs on the right side and `a[j]` on the left side; both are on the wrong side. After the swap both are on the correct side, and the scans resume where they left off.

- **Lines 11–12.** When the loop exits, `a[i]` is the first item of the right side. The pivot is swapped with it and `i` is returned.

The partition invariant is described by the four regions in the figure. At every step of the loop, the items to the left of `i` (`a[L..i-1]`) are not greater than *v*, and the items to the right of `j` (`a[j+1..R-1]`) are not less than *v*. Between them lies a region not yet scanned. Each swap shrinks the unscanned region by two items. When the indices cross, this region is empty.

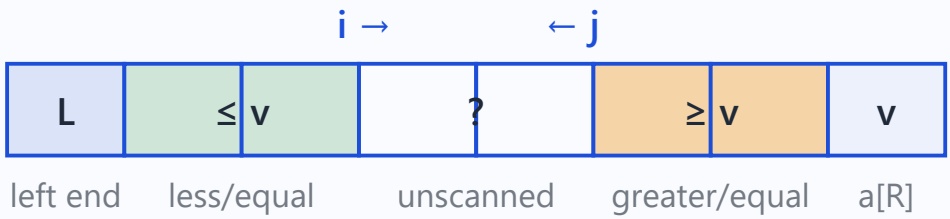


Figure 20.1: The partition invariant. Items less than or equal to *v* on the left, items greater than or equal to *v* on the right, the not-yet-scanned region in the middle, and the pivot *v = a[R]* at the far right.

Having both scans stop on equal keys is a deliberate choice. Consider an array whose items are all equal. If the scans did not stop on equality, `i` would always run to the end. Every split would produce parts of $n - 1$ and 0 items, and the running time would reach n^2. When the scans stop on equal keys, seemingly needless swaps are made, but the indices meet in the middle.

The cost of `partition` on a range of $n = R - L + 1$ items is $\Theta(n)$. `i` moves only right and `j` only left. Each item is examined once until they meet; at the crossing, at most a few items are examined a second time. The number of comparisons is n or $n + 1$. The extra memory is a few variables: $O(1)$.

20.2.3 Code trace: `partition(a, 0, 9)`

Let the array be *[37, 15, 24, 33, 49, 11, 51, 77, 72, 48]*. The pivot is *v = a[9] = 48*. The table gives, at each step, where the indices stop and the state of the array; the figure shows four key moments.

Step	Line	`i`	`j`	Event	Array
0	2	−1	9	start	`37 15 24 33 49 11 51 77 72 48`
1–4	4	0…3	9	37, 15, 24, 33 < 48: `i` advances	same
5	4	4	9	*a[4] = 49 ≥ 48*: `i` stops	same
6–8	6	4	8…6	72, 77, 51 > 48: `j` moves left	same
9	6	4	5	*a[5] = 11 ≤ 48*: `j` stops	same
10	9	4	5	*i < j*: swap	`37 15 24 33 11 49 51 77 72 48`
11	4	5	5	*a[5] = 49 ≥ 48*: `i` stops	same
12	6	5	4	*a[4] = 11 ≤ 48*: `j` stops	same
13	8	5	4	*i ≥ j*: leave the loop	same

Step	Line	`i`	`j`	Event	Array
14	11–12	5	4	pivot goes to `a[5]`; returns 5	`37 15 24 33 11 48 51 77 72 49`

Table 20.1: Trace of `partition(a, 0, 9)`. Pivot $v = 48$.

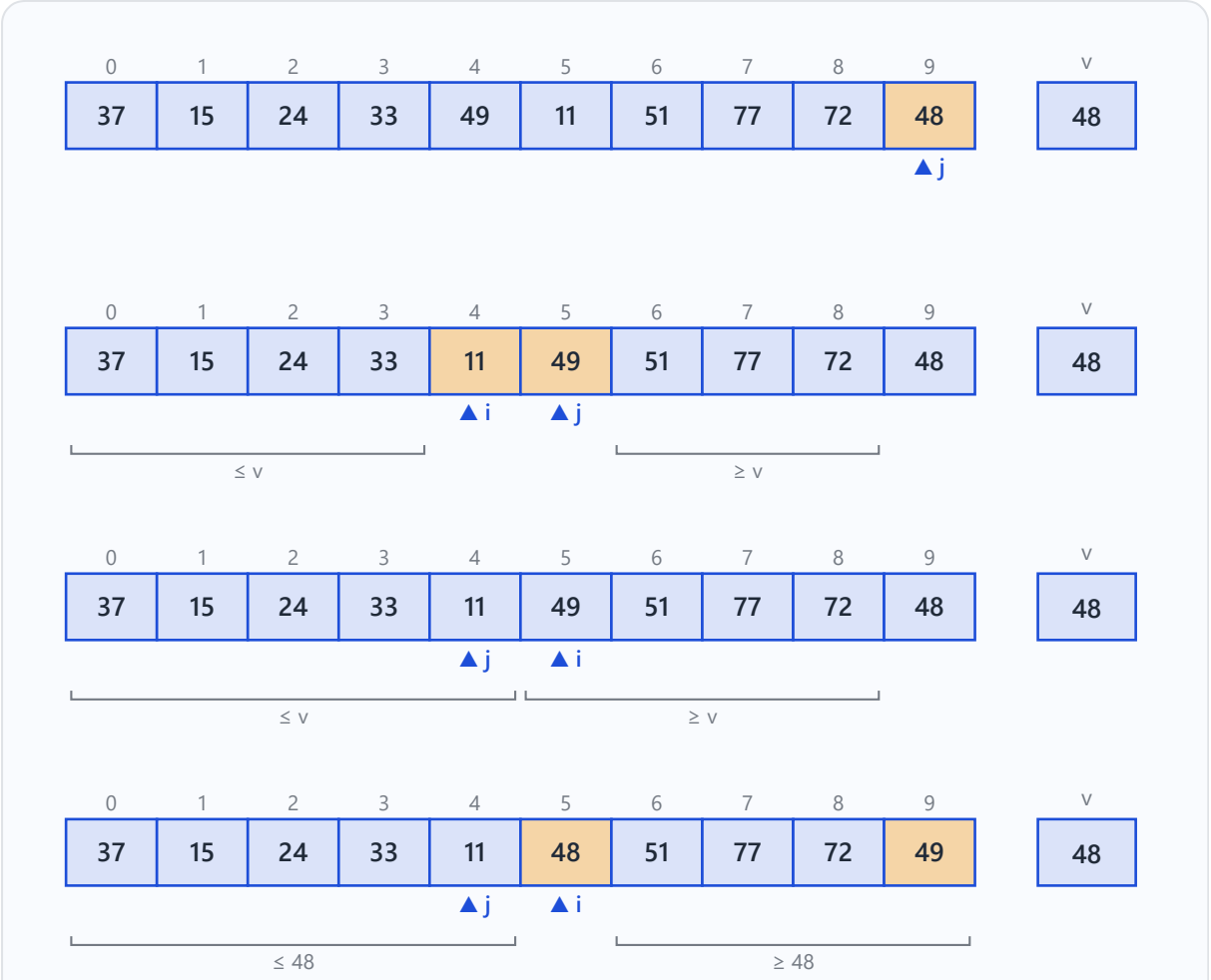


Figure 20.2: Four moments of the `partition(a, 0, 9)` trace, top to bottom: the start ($i = -1, j = 9$); the swap of 49 and 11; the crossing of the indices ($i = 5, j = 4$); the pivot moving to position 5. Orange cells are the items exchanged in that step; the brackets underneath mark the $\le v$ and $\ge v$ regions.

In the final state, 48 has 37, 15, 24, 33, 11 on its left, all smaller than 48. On its right are 51, 77, 72, 49, all larger. Note that neither side is sorted internally. Partitioning fixes only the pivot's position: if the data were sorted, 48 would be exactly at position 5. This trace made $n + 1 = 11$ comparisons: the `i` scan looked at 6 items and the `j` scan at 5. Positions 4 and 5 were examined by both scans; the extra comparison comes from the crossing.

Common mistake

Deleting the line `if (j == L) break;` in the `j` loop on the reasoning "the `i` loop has no bounds check either, so none is needed here." The two scans are not symmetric. On the right the pivot stops `i`, but on the left nothing stops `j`. If the pivot is the smallest item in the

range, `j` runs to `a[L-1]`, even into memory before the start of the array. This is undefined behavior; the program sometimes seems to work and sometimes crashes.

20.2.4 Code trace: `quicksort1(a, 0, 9)`

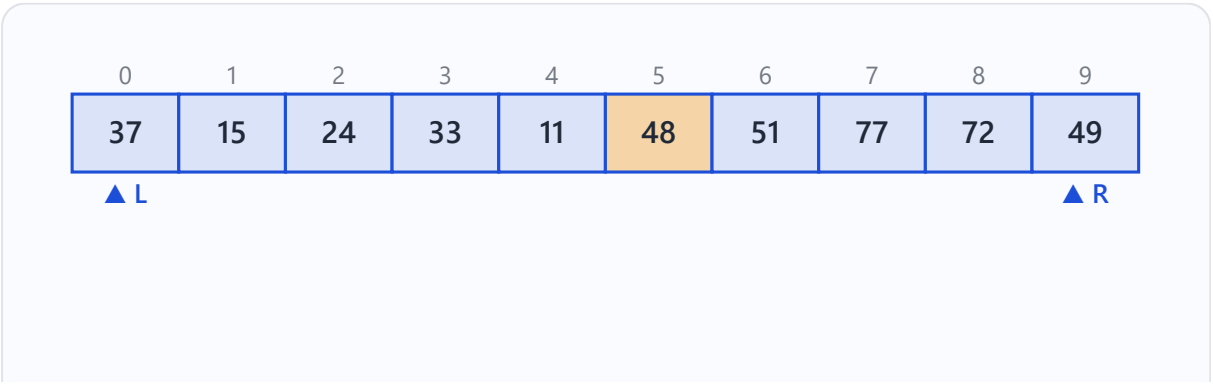
Now let the same array be sorted from start to finish with `quicksort1`. The table gives each `partition` call, the position it returns, and the array afterwards. The recursion finishes the left part completely before moving on to the right part, so the call stack gets deep along the left parts.

Call	Pivot	Position	Next calls	Array (after)
`partition(0, 9)`	48	5	(0, 4), (6, 9)	`37 15 24 33 11 48 51 77 72 49`
`partition(0, 4)`	11	0	(0, −1), (1, 4)	`11 15 24 33 37 48 51 77 72 49`
`partition(1, 4)`	37	4	(1, 3), (5, 4)	same
`partition(1, 3)`	33	3	(1, 2), (4, 3)	same
`partition(1, 2)`	24	2	(1, 1), (3, 2)	same
`partition(6, 9)`	49	6	(6, 5), (7, 9)	`11 15 24 33 37 48 49 77 72 51`
`partition(7, 9)`	51	7	(7, 6), (8, 9)	`11 15 24 33 37 48 49 51 72 77`
`partition(8, 9)`	77	9	(8, 8), (10, 9)	same

Table 20.2: The `partition` calls made during `quicksort1(a, 0, 9)`. Calls on empty or single-item ranges return immediately and are not shown.

In `partition(0, 4)` the pivot, 11, is the smallest in the range. `i` stops at once at `a[0] = 37`. `j`, however, moves left past 33, 24, and 15 and stops at 0 thanks to the `j == L` check. The indices cross, the pivot lands at position 0, and 37 moves to the end. The left part is empty. This is the bounds check in the `j` loop at work.

In total 17 `quicksort1` calls are made. Eight of them call `partition`, and each puts one pivot in its final place. The remaining nine are called with empty or single-item ranges and return immediately: *(0, −1)*, *(1, 1)*, *(3, 2)*, *(4, 3)*, *(5, 4)*, *(6, 5)*, *(7, 6)*, *(8, 8)*, *(10, 9)*. Because every `partition` call spawns two subcalls, the number of leaves is one more than the number of internal nodes: *8 + 9 = 17*. At the deepest moment the stack holds six `quicksort1` frames above `main`: (0, 9), (0, 4), (1, 4), (1, 3), (1, 2), and (1, 1).



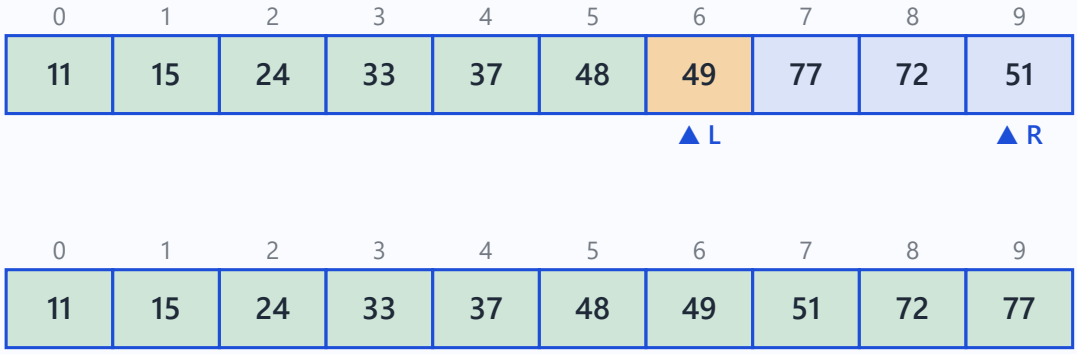


Figure 20.3: Three moments of `quicksort1(a, 0, 9)`, top to bottom: after the first partition, with 48 in place; after the left part is done and `partition(6, 9)` has run; the last empty call (10, 9). Green cells are items that have settled in their final positions.

20.2.5 Why is it fast?

In bubble sort and selection sort too, the correct position of one item is found at every step. The difference is this: they put one item in place but do nothing to shrink the problem. The subproblem of the next step has $n - 1$ items. Quicksort, while placing the pivot, also splits the problem into two smaller parts. With a good split, each part has about $n/2$ items. At the next level, four parts have $n/4$ items each, and so on. After $\lg n$ levels the parts shrink to single items. The total partitioning work at each level is around n, so the overall work is of order $n \lg n$.

All the quicksort versions in this note use the same `partition`. Code this short cannot be grasped by reading it once; it should be traced on several different arrays and rewritten from its logic.

20.2.6 Algorithms and computers

Students often ask: where did Hoare try this algorithm in the 1960s, and how did he test it? Where does the word “algorithm” come from; did al-Khwarizmi have a laptop in the 9th century?

The answer lies in the concept of an algorithm itself. An algorithm is a precise method that someone who knows nothing about the subject can carry out step by step and still reach the correct result; computers do not appear in the definition. Computers did exist in the 1960s, but the correctness and cost of an algorithm are proved on paper, and publishing the proof in a paper is enough. That is why this course is independent of any computer.

The origin of the word confirms this. Al-Khwarizmi (780–850) was a mathematician and astronomer who worked at the House of Wisdom in Baghdad. He wrote *al-Kitab al-Mukhtasar fi Hisab al-Jabr wa-l-Muqabala*, which gives a systematic solution of linear and quadratic equations. The word “algebra” comes from the name of the method *al-jabr* in this book. “Algorithm” derives from *Algoritmi*, the Latinized form of al-Khwarizmi's name. Al-Khwarizmi is also known for the works that introduced the decimal positional number system to the West.

20.3 Analysis of quicksort

The only thing that determines quicksort's running time is how `partition` divides the range. Each `partition` call does work proportional to the length of its range. If the splits are balanced, the recursion tree stays shallow; if they are unbalanced, the tree turns into a chain. In this section cost is measured by the number of comparisons, because comparison is the real work of sorting and the number of swaps is proportional to it. C_n denotes the number of comparisons made while sorting an array of n items.

20.3.1 Worst case: a sorted array

Suppose the array is already sorted. The pivot `a[R]` is the largest item of the range every time. The `i` scan passes the whole range and stops at `R`; `j` stops at the first item it looks at. The pivot stays in place, the left part has $n - 1$ items and the right part is empty. The next call again takes the largest item as pivot and again continues with one item fewer. The total work is

$$n + (n - 1) + \ldots + 2 + 1 = n(n + 1)/2 = O(n^2)$$

The exact count can also be computed. On a range of size s, `partition` makes exactly $s + 1$ comparisons: s for the `i` scan and 1 for the `j` scan. The $n - 1$ ranges of sizes $n, n - 1, \ldots, 2$ are partitioned. For $n = 10$ the total is $11 + 10 + \ldots + 3 = 63$ comparisons.

Moreover, the recursion depth reaches $n - 1$; large sorted arrays can cause a **stack overflow** on systems with small stacks. A sorted array of 100,000 items took 0.95 seconds with `-O2`, while a random array of the same size took 0.008 seconds (times vary by machine). A reverse-sorted array is just as bad.

20.3.2 Best case: splitting exactly in the middle

In the best case the pivot is exactly the median of the range every time. `i` and `j` together advance n steps, and two subproblems of $n/2$ items each remain:

$$C_n = 2C_{n/2} + n,\ C_1 = 0$$

The recurrence is expanded by substituting it into itself. Since $C_{n/2} = 2C_{n/4} + n/2$, we get $C_n = 2(2C_{n/4} + n/2) + n = 4C_{n/4} + 2n$. One more expansion gives $8C_{n/8} + 3n$. After k steps

$$C_n = 2^k\, C_{n/2^k} + k \cdot n$$

Taking $n = 2^k$, that is $k = \lg n$, gives $C_{n/2^k} = C_1 = 0$ and so $C_n = n \lg n$.

The lower bound for comparison sorting was $\lceil \lg n! \rceil \approx n \lg n - 1.44\, n$. The leading coefficient is again 1; the gap is only about $1.44\, n$. The real question is behavior on a random array: the answer to "what is fast about this?" lies in the average case.

20.3.3 Counting: sorted and random input

To make the difference between the two extremes concrete, the comparisons in `partition` were counted. The table gives the number of comparisons and the recursion depth for sorted and random input. Depth is the largest number of nested, waiting `quicksort1` frames that are calling `partition`.

n	Sorted: comparisons	Depth	Random: comparisons	Depth	$n \lg n$	Random average
10	63	9	36	5	33	35.4
100	5,148	99	721	9	664	765.6
1000	501,498	999	12,831	25	9,966	12198.3

Table 20.3: Comparison count and recursion depth in `quicksort1`. The random columns are a single trial from the slide deck (Python simulation, seed 20); the last column is the average over many random arrays.

On sorted input the number of comparisons is $(n + 1)(n + 2)/2 - 3$, about $n^2/2$, and the depth is $n - 1$. At $n = 1000$, half a million comparisons are made. On random input the count stays of order $n \lg n$ and the depth is only a few times $\lg n$. For two inputs of the same size the work differs by a factor of forty. Sorted or nearly sorted data is not rare in real life.

20.3.4 Average case: setting up the recurrence

Average-case analysis rests on mathematics, not intuition. When a coin is tossed a million times, the fraction of heads approaches ½; average-case analysis pins down in the same way where the outcomes concentrate. Saying "the probabilities are equal as far as I am concerned" is not enough; the equality has to be proved.

Let all $n!$ orderings of the input be equally likely and the keys distinct. The pivot is `a[R]`. In a random ordering, the rank of the rightmost item in sorted order takes each of the values $1, 2, \ldots, n$ with equal probability $1/n$. If the pivot has rank k, `partition` places it at position k, leaving two subarrays of $k - 1$ and $n - k$ items. One more thing is required: the subarrays must also be random. `partition` compares items only with the pivot, never with each other. So the orderings inside each part are again equally likely, and the parts obey the same analysis.

`partition` makes $n + 1$ comparisons against the pivot. Taking this count as the model, the average number of comparisons satisfies the following recurrence:

$$C_n = (n + 1) + (1/n) \sum_{k=1}^{n} (C_{k-1} + C_{n-k}),\ C_0 = C_1 = 0$$

The first term is the partitioning work. The sum is the cost of the two subproblems for each value of k; the factor $1/n$ comes from those values being equally likely. The two sums contain the same terms. For $k = 1, \ldots, n$ the terms C_{k-1} are $C_0, C_1, \ldots, C_{n-1}$. The terms C_{n-k} are the same values, only in reverse order. Therefore

$$\Sigma_{k=1}^{n} (C_{k-1} + C_{n-k}) = 2 \Sigma_{k=1}^{n} C_{k-1}$$

can be written. Multiplying both sides of the recurrence by n removes the fraction:

$$n \cdot C_n = n(n + 1) + 2 \Sigma_{k=1}^{n} C_{k-1} \quad (1)$$

The real `partition` makes n or $n + 1$ comparisons in a comparison scan; if the indices meet at the same position it counts one fewer. In an experiment with 20,000 random arrays, about 90% of the calls made $n + 1$ comparisons and the rest made n. So the measured averages come out slightly below C_n (last column of the table). The recurrence is a slightly pessimistic model; the asymptotic result does not change.

20.3.5 The subtraction trick and the solution

The sum in recurrence (1) contains all earlier values for every n. Recurrences of this kind are usually solved by substitution, and Horowitz's book does it that way. There is a shorter route: eliminate the sum. In (1), replace n by $n - 1$:

$$(n - 1) \cdot C_{n-1} = (n - 1)n + 2 \Sigma_{k=1}^{n-1} C_{k-1} \quad (2)$$

Subtracting (2) from (1), the common terms of the two sums cancel, leaving only $2C_{n-1}$. The difference of the first terms on the right is $n(n + 1) - n(n - 1) = 2n$:

$$n \cdot C_n - (n - 1) \cdot C_{n-1} = 2n + 2C_{n-1} \Rightarrow n \cdot C_n = (n + 1) \cdot C_{n-1} + 2n$$

Now the recurrence depends only on the previous term. Divide both sides by $n(n + 1)$:

$$C_n/(n + 1) = C_{n-1}/n + 2/(n + 1)$$

The beauty of this form is that the left side and the first term on the right have the same shape. Let $D_n = C_n/(n + 1)$; then $D_n = D_{n-1} + 2/(n + 1)$. This unrolls at once by iteration: $D_n = D_{n-1} + 2/(n + 1) = D_{n-2} + 2/n + 2/(n + 1) = \ldots$. The subtraction step requires (2) to hold for $n - 1 \geq 2$. So the expansion is stopped at $C_2 = 3$, where $D_2 = 1$:

$$C_n/(n + 1) = 1 + 2(1/4 + 1/5 + \ldots + 1/(n + 1)) \; (n \geq 2)$$

The sum in parentheses is the harmonic number $H_{n+1} = 1 + 1/2 + \ldots + 1/(n+1)$ with its first three terms removed. The dominant part is $2(n+1)(1/4 + \ldots + 1/(n+1))$, which is less than $2(n+1)\ln n$. Since $H_n \approx \ln n + 0.577$,

$$C_n \approx 2n \ln n = 2 \ln 2 \cdot n \lg n \approx 1.39\, n \lg n$$

follows. The last step uses the identity $\ln n = \ln 2 \cdot \lg n$; and $2 \ln 2 \approx 1.386$.

Theorem 20.1 (Average case of quicksort)

If every ordering of n items with distinct keys is equally likely and `partition` makes $n + 1$ comparisons, the average number of comparisons of `quicksort1` is $C_n = 2(n+1)H_{n+1} - 8(n+1)/3$ $(n \geq 2)$. Consequently $C_n \approx 2n \ln n \approx 1.39\, n \lg n$ and $C_n = \Theta(n \lg n)$.

Proof. The closed form follows from the expansion above: $C_n = (n+1)(1 + 2(H_{n+1} - 1 - 1/2 - 1/3)) = 2(n+1)H_{n+1} - (n+1)(2 \cdot 11/6 - 1)$, and $2 \cdot 11/6 - 1 = 8/3$. For $n = 2$ we get $3 \cdot (2 \cdot 11/6) - 8 = 3$, which agrees with $C_2 = 3 + (2/2)(0 + 0) = 3$ from the recurrence. Since $H_{n+1} = \ln n + O(1)$, the leading term is $2n \ln n$.

The recurrence was evaluated exactly in Python and compared with the formula. The table gives the result for three sizes. The ratio approaches 1 very slowly. The reason is the lower-order terms of the closed form, of the type $-cn$, whose importance fades only gradually next to $n \lg n$.

n	C_n (from the recurrence)	$1.39\, n \lg n$	Ratio
10	37.1	46.2	0.80
100	780.5	923.5	0.85
1000	12318.6	13852.4	0.89

Table 20.4: Average number of comparisons. C_n was computed exactly from the recurrence.

On random input, quicksort makes only about 39% more comparisons than in its best case; the probability of the n^2 case is tiny. Its inner loop is also very short: an increment, a comparison, and a branch, with items read from memory in order.

20.4 Improving quicksort

`quicksort1` has two weaknesses. First, in the worst case the recursion depth reaches n and can exhaust the stack. Second, sorted or reverse-sorted input drags it to n^2. In addition, on small subarrays the cost of recursion exceeds the useful work. This section examines three improvements in turn: a non-recursive version with an explicit stack, choosing the pivot as the median of three items, and leaving small subarrays to insertion sort.

20.4.1 The non-recursive version and the explicit stack

A recursive routine keeps its pending work on the call stack; every call opens a **stack frame** carrying its parameters, local variables, and return address. Each pending job of `quicksort1` is a pair *(L, R)*. If these pairs are pushed onto a stack implemented with an array, recursion is unnecessary; the call overhead disappears and the order in which jobs are done comes under control.

```
#define YMAX 64     /* > 2·lg n */
static int stack[YMAX], top;
void stack_init(void) { top = 0; }
int stack_empty(void) { return top == 0; }
void push(int x) { stack[top++] = x; }
int pop(void) { return stack[--top]; }
#define push_range(A, B) \
  do { push(B); push(A); } while (0)
```

The stack is a `static` `int` array of size `YMAX`. `top` is the index of the first free cell; `top == 0` means the stack is empty. The `static` keyword makes the array and `top` visible only within this source file and keeps both alive for the whole program. `push` writes the value to `stack[top]` and increments `top`. `pop` first decrements `top`, then returns that cell. Both operations are *O(1)*.

The macro `push_range(A, B)` pushes a pair. It pushes `B` first, then `A`. Since the first item popped is the last one pushed, the first item popped is `A`. When `push_range(L, R)` is written in `quicksort2`, the pop order is `L`, then `R`. The body of the macro is again wrapped in `do { … } while (0)` so that it can be used as a single statement in `if`–`else` branches. The backslash (`\`) at the end of a line tells the preprocessor that the macro definition continues on the next line.

The stack does no overflow checking; its size was chosen so that no input can exceed it. By Theorem 20.2, the stack holds at most $\lfloor \lg n \rfloor + 1$ pairs, that is $2(\lfloor \lg n \rfloor + 1)$ `int`s. For $n < 2^{31}$ this does not exceed 62.

```
void quicksort2(int a[], int L, int R) {
  stack_init(); push_range(L, R);
  while (!stack_empty()) {
    L = pop(); R = pop();
    if (R <= L) continue;
    int i = partition(a, L, R);
    if (i - L > R - i) {       /* larger part first */
      push_range(L, i - 1); push_range(i + 1, R);
    } else {
      push_range(i + 1, R); push_range(L, i - 1);
    }
  }
}
```

- **Line 2.** The stack is emptied and the first problem, the whole range, is pushed.

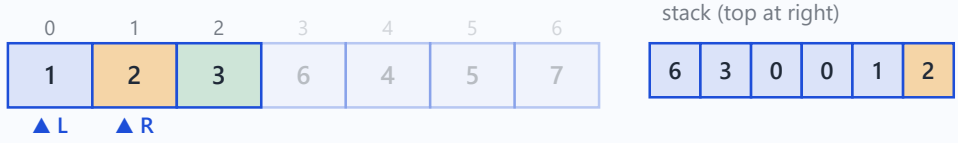
- **Lines 3–5.** Until the stack is empty, a pair is popped. If the range is empty or has one item, `continue` moves to the next pair. These lines correspond to the base case of `quicksort1`.
- **Line 6.** The range is partitioned. The left part is `a[L..i-1]` and the right part is `a[i+1..R]`; their sizes are $i - L$ and $R - i$.
- **Lines 7–11.** The larger part is pushed first and the smaller part last. Because the stack gives back the last item first, the next round always processes the smaller part. If the sizes are equal, the `else` branch runs and the left part stays on top.

20.4.2 Code trace: `quicksort2(a, 0, 6)`

Let the array be *[4, 7, 2, 6, 1, 5, 3]*. The table gives the contents of the stack after each round, from bottom to top. Pairs sit on the stack in the order *R, L*; the top two numbers are the *L* and *R* of the next job.

Round	Popped	Event	Array (after)	Stack (after)
0	—	first problem is pushed	`4 7 2 6 1 5 3`	`[6, 0]`
1	(0, 6)	pivot 3 → 2; right (3, 6) is larger, so it goes first; then left (0, 1)	`1 2 3 6 4 5 7`	`[6, 3, 1, 0]`
2	(0, 1)	pivot 2 → 1; left (0, 0) is larger, so it goes first; then right (2, 1)	same	`[6, 3, 0, 0, 1, 2]`
3	(2, 1)	empty range: `continue`	same	`[6, 3, 0, 0]`
4	(0, 0)	single item: `continue`	same	`[6, 3]`
5	(3, 6)	pivot 7 → 6; left (3, 5) is larger, so it goes first; then right (7, 6)	same	`[5, 3, 6, 7]`
6	(7, 6)	empty range: `continue`	same	`[5, 3]`
7	(3, 5)	pivot 5 → 4; parts are equal, right (5, 5) first, then left (3, 3)	`1 2 3 4 5 6 7`	`[5, 5, 3, 3]`
8	(3, 3)	single item: `continue`	same	`[5, 5]`
9	(5, 5)	single item: `continue`	same	`[]`
10	—	stack empty: the loop ends	sorted	`[]`

Table 20.5: Trace of `quicksort2(a, 0, 6)`. The stack is written from bottom to top.



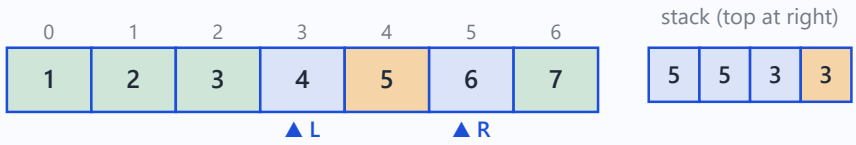


Figure 20.4: Two moments of the `quicksort2` trace. Top: the end of round 2, with at most three pairs waiting on the stack. Bottom: the end of round 7, with the array sorted and only two single-item jobs left on the stack. The top of the stack is on the right.

At most 3 pairs, that is 6 `int`s, were waiting on the stack. Single-item and empty ranges are pushed and popped before being discarded; checking for them beforehand would save a few operations but lengthen the code.

20.4.3 Why does the stack stay small?

Theorem 20.2 (Size of the explicit stack)

While `quicksort2` sorts an array of n items, the stack holds at most $\lfloor \lg n \rfloor + 1$ pairs at any one time. The extra memory is $O(\lg n)$.

Proof. When a range is partitioned, the larger and the smaller part are pushed. The smaller part has at most $\lfloor (s - 1)/2 \rfloor$ items, because the two parts total $s - 1$. The smaller part is popped immediately. Every pair pushed between that moment and the popping of the larger part comes from inside the smaller part. So at any moment the larger parts waiting on the stack are remnants of the partitioning of nested ranges: $S_1 \supset S_2 \supset \ldots \supset S_k$. Each S_{i+1} lies inside the smaller part of S_i, so $|S_{i+1}| < |S_i|/2$. Since $|S_1| \le n$, we get $|S_k| < n/2^{k-1}$. A range that gets partitioned contains at least two items: $2 \le |S_k| < n/2^{k-1}$, that is $2^k < n$ and $k \le \lfloor \lg n \rfloor$. In addition to these k larger parts, at most one smaller part waits on the stack.

The same result follows from a memory recurrence: the smaller part has at most half the size and each level adds two `int`s to the stack; $T_n = T_{n/2} + 2$ gives $T_n = 2 \lg n$, as in binary search. The table compares the bound with measurements. On sorted input the stack stays almost empty, because the smaller part is empty.

n	$\lfloor \lg n \rfloor + 1$	Random input	Sorted input
7	3	3	2
100	7	6	2
1000	10	9	2
10,000	14	10	2

n	*⌊lg n⌋ + 1*	Random input	Sorted input
1,048,576	21	15	—

Table 20.6: The largest number of pairs waiting on the stack at one time in `quicksort2`. The random column is the maximum over 2000 arrays for $n \le 1000$ and over 5 arrays for larger n.

Even though the stack stays small on sorted input, the time is still n^2; `quicksort2` sorted 100,000 sorted items in 0.94 seconds. The non-recursive version solves the memory problem but not the time problem. The time problem is solved by pivot selection.

Common mistake

Pushing the parts without looking at their sizes, for example always pushing the left part first and the right part second. The code still sorts correctly, but the *O(lg n)* bound on the stack is lost. On a bad input, of the order of n pairs pile up on the stack; since `YMAX` is 64, the array overflows and the program overwrites its own variables.

20.4.4 Median-of-three and the small-subarray cutoff

Quicksort performs worst on sorted data. Insertion sort, by contrast, performs best on exactly sorted data. The improved version combines these two observations:

```
#define M 10    /* leave subarrays shorter than this */
void quicksort3(int a[], int L, int R) {
    if (R - L <= M) return;
    SWAP(a[(L + R) / 2], a[R - 1]);
    if (a[R-1] < a[L]) SWAP(a[R-1], a[L]); // a[L] <= a[R-1]
    if (a[R] < a[L]) SWAP(a[R], a[L]);     // a[L] <= a[R]
    if (a[R] < a[R-1]) SWAP(a[R], a[R-1]); // a[R-1] <= a[R]
    int i = partition(a, L + 1, R - 1);
    quicksort3(a, L, i - 1);
    quicksort3(a, i + 1, R);
}
void hybrid_sort(int a[], int L, int R) {
    quicksort3(a, L, R);
    insertion(a, L, R);   /* leftover small parts */
}
```

Median-of-three, or median-3 for short, chooses the pivot as the median of three items instead of a single item. These three items are the first item of the range, the middle item, and the last item. Line by line:

- **Line 3.** If the range has at most $M + 1$ items ($R - L \le M$), the routine returns without doing anything. Such parts are left unsorted.

- **Line 4.** The middle item is moved to `a[R-1]`. The three candidates are now `a[L]`, `a[R-1]`, and `a[R]`.
- **Lines 5–7.** Three "compare, swap if needed" steps put these three items in order. At the end $a[L] \le a[R-1] \le a[R]$ holds and the median is in `a[R-1]`.
- **Line 8.** Partitioning is done only on `a[L+1..R-1]`. `partition` takes the last item of this range, the median, as the pivot.
- **Lines 9–10.** The two sides of the pivot are processed recursively.

In the three swap lines the order of the arguments matters. The pattern `if (less(A, B)) SWAP(A, B)` says "swap if $A < B$." After the swap, the item in `B` is the smaller of the two and the item in `A` the larger. The first line of the code, `if (a[R-1] < a[L]) SWAP(a[R-1], a[L]);`, puts the smaller item into `a[L]`. If the arguments were given in the order `(a[L], a[R-1])`, the larger item would go to `a[L]` and partitioning would break. This wrong version was tried on 1000 random arrays; in every one, at least one partition left an item larger than the pivot on its left.

Median-3 has two benefits. On sorted and reverse-sorted input the pivot comes from the middle of the range and the n^2 case disappears. A sorted array of 100,000 items took 0.95 seconds with `quicksort1` and 0.001 seconds with `hybrid_sort`. In addition, the end items become **natural sentinels**: since $a[L] \le pivot \le a[R]$, the `j` scan stops at `a[L]` at the latest.

The second improvement is the cutoff M. On a range of a few items, the fixed cost of the call and its setup exceeds the real work, so `quicksort3` leaves parts with $R - L \le M$ untouched. The array stays "nearly sorted": every item is out of place only inside its own small part, and the parts are in the correct order relative to each other. At the end, `hybrid_sort` applies a single insertion-sort pass:

```
void insertion(int a[], int L, int R) {
    for (int i = L + 1; i <= R; i++) {
        int v = a[i];      // item being inserted
        int j = i - 1;     // a[L..i-1] is sorted
        while (j >= L && a[j] > v) {
            a[j + 1] = a[j]; // shift right
            j--;
        }
        a[j + 1] = v;
    }
}
```

The cost of insertion sort depends on how many positions the items shift. Here no item moves outside its own part, so the final pass runs in $O(n \cdot M)$ time, linear for a fixed M.

20.4.5 Code trace: `quicksort3(a, 0, 8)`

To keep the trace short, take $M = 3$. The array is $[8, 3, 5, 1, 9, 2, 7, 6, 4]$. The table gives the setup steps and the recursion.

Line	Event	Array (after)
3	$R - L = 8 > 3$: the part is large and will be partitioned	`8 3 5 1 9 2 7 6 4`
4	middle item `a[4] = 9` is swapped with `a[7] = 6`	`8 3 5 1 6 2 7 9 4`
5	$a[7] = 9 \geq a[0] = 8$: no swap	same
6	$a[8] = 4 < a[0] = 8$: swap	`4 3 5 1 6 2 7 9 8`
7	$a[8] = 8 < a[7] = 9$: swap	`4 3 5 1 6 2 7 8 9`
8	`partition(a, 1, 7)` : pivot 8, position 7	same
9	left `a[0..6]` : $R - L = 6 > 3$; after median-3, `a[0..6] = 1 3 5 2 6 4 7`	`1 3 5 2 6 4 7 8 9`
9 → 8	`partition(a, 1, 5)` : pivot 4, position 3	`1 3 2 4 6 5 7 8 9`
9 → 9, 10	`a[0..2]` and `a[4..6]` : $R - L = 2 \leq 3$, they return; then on line 10 the right part `a[8..8]` ($R - L = 0 \leq 3$) returns	same
`hybrid_sort`	final insertion-sort pass	`1 2 3 4 5 6 7 8 9`

Table 20.7: Trace of `quicksort3(a, 0, 8)` , $M = 3$.

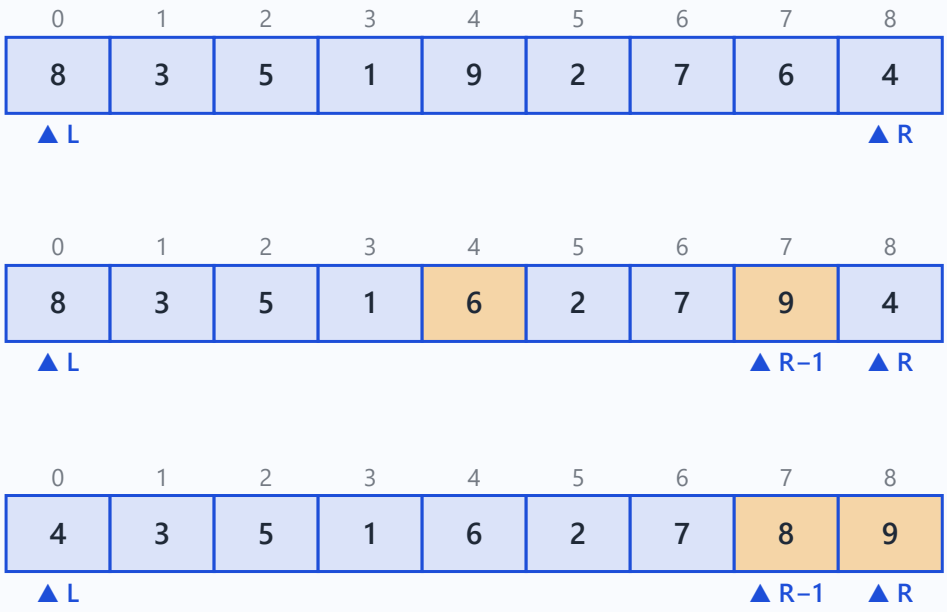


Figure 20.5: The median-3 setup, top to bottom: the start; after the middle item has been moved to `a[R-1]` ; after the three comparisons, $a[L] = 4 \leq a[R-1] = 8 \leq a[R] = 9$.

In this trace the first pivot is unlucky. The three candidates are 8, 9, and 4; their median, 8, is the second largest item of the array. The split is 7 to 1. Median-3 does not rule out a bad split entirely; it only reduces its probability. In the left part the candidates are 4, 1, and 7. This time the median, 4, falls exactly in the middle of the part. When `quicksort3` finishes, the array is *[1, 3, 2, 4, 6, 5, 7, 8, 9]*: the parts are sorted relative to each other, but 3 and 2, and 6 and 5, are swapped within their own parts. The final insertion pass fixes these two pairs with two shifts.

With *M = 10*, on this nine-item array `quicksort3` returns on the first line and insertion sort does all the work. Both versions were tested on 3000 random arrays with repeated keys.

20.4.6 The effect and limits of the cutoff

Sources report that cutoffs in the range $5 \le M \le 25$ reduce the running time by about 10%. The gain depends on the machine and is small; many repetitions are needed to measure it. The table gives the time for sorting one million random items with `hybrid_sort` for different values of *M*. *M = 1* is the case where the cutoff is practically switched off: since median-3 needs at least three items, this is the smallest meaningful value.

M	1	5	10	15	25	50
Time (s)	0.105	0.092	0.089	0.101	0.089	0.093

Table 20.8: Time of `hybrid_sort` on *n = 1,000,000* random items (`zig cc -O2` , x86-64; median of three measurements, each measurement the average of five sorts). Times vary by machine, and individual measurements fluctuate by ±15%.

The cutoff cut the time by 10–15%, and no clear difference appeared among values between 5 and 50. Because the differences are close to noise, the measurement was repeated and the median taken.

Median-3 has a limit. It rescues sorted and reverse-sorted input, but not every input. Inputs constructed on purpose so that the median candidates always come out small still take n^2. On random data this probability is negligible; but in systems where an attacker can choose the input, such as a network server, it is a weakness. The definitive fix is a hybrid method with a worst-case guarantee: parts whose recursion gets too deep are sorted with heap sort.

It is also a new situation that a measurement like this takes a few minutes today. In the past the program was transferred onto punched cards, and a person got a compilation turn once a day, twice at most. One may suppose that mandatory internships were introduced in that era of scarcity to make up for the lack of practice. Today dozens of compilations a day are possible. The aim of this course is likewise to train engineers who think like scientists: not merely applying ready-made patterns, but proving an algorithm, measuring it, and, when necessary, constructing a new one.

20.5 The C library and hybrid sorts

20.5.1 `qsort`

The standard C library provides a general-purpose sorting routine in the header `<stdlib.h>`:

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

The parameters are the address of the array, the number of items, the size of one item in bytes, and the address of the comparison function. `qsort` does not know the item type; it sees the array as blocks of `size` bytes. For this reason the first parameter is a **void pointer**: a pointer that can carry the address of any object but carries no type information. A `void *` cannot be dereferenced directly; it is first converted to the real type.

The fourth parameter is a **function pointer**: the address of a function that takes two `const void *` and returns an `int`. In C, a function name written without parentheses yields the address of the function. When `qsort` compares two items, it calls this function with the addresses of the items; by changing the comparator, any type can be sorted.

```
int ascending(const void *p, const void *q) {
    int x = *(const int *)p;
    int y = *(const int *)q;
    return (x > y) - (x < y);  // -1, 0, +1
}
int main(void) {
    int a[] = {37, 15, 24, 33, 49};
    size_t n = sizeof a / sizeof a[0];
    qsort(a, n, sizeof a[0], ascending);
    for (size_t i = 0; i < n; i++)
        printf("%d ", a[i]);
    return 0;
}
```

The program is compiled with the headers `<stdio.h>` and `<stdlib.h>` and prints `15 24 33 37 49`.

- `ascending`. The incoming `void` pointers are converted to `const int *` and dereferenced. `const` states that the comparison function will not modify the items. The return value is `(x > y) - (x < y)`. In C, a comparison operator yields 1 if true and 0 if false. If $x > y$ the result is $1 - 0 = 1$, if $x < y$ it is $0 - 1 = -1$, and if they are equal it is 0.
- `sizeof a / sizeof a[0]`. The total size of the array is divided by the size of one item. With a 4-byte `int` this gives $20 / 4 = 5$. This expression gives the correct result only where the array is defined. When the array is passed to a function as a parameter, it decays to a pointer and `sizeof`

gives the size of the pointer.

- `qsort(a, n, sizeof a[0], ascending)`. `ascending` is written without parentheses; it is not called, its address is passed. The item size is written as `sizeof a[0]` rather than `sizeof(int)`. If the type of the array changes, this expression stays correct automatically.

20.5.2 Code trace: `qsort` and the comparison function

Step	Event	Values
1	array ready; `n = 20 / 4 = 5`	`a = [37, 15, 24, 33, 49]`
2	`qsort(a, 5, 4, ascending)` is called	`main` and `qsort` on the stack
3	`ascending(&37, &15)`	*x = 37, y = 15*, returns 1: the first item must come later
4	`ascending(&15, &24)`	returns −1: the first item must come first
5	`ascending(&24, &24)`	returns 0: equal, relative order not guaranteed
6	`qsort` returns	`a = [15, 24, 33, 37, 49]`
7–11	the loop prints the items	output: `15 24 33 37 49`

Table 20.9: Trace of the sample program. The `ascending` rows are example calls; the standard does not specify which pairs `qsort` compares or in what order.



Figure 20.6: The array before (left) and after (right) the `qsort` call.

The comparison function must obey a contract. It returns a negative value if the first item must come before the second, a positive value if it must come after, and zero if they are equal. The magnitude of the return value is irrelevant; only its sign is used. The function must also be consistent: if it says $a < b$ and $b < c$, it must say $a < c$. It must give the same answer every time for the same two items and must not modify the items. If any of these rules is broken, the behavior of `qsort` is undefined: the array may be sorted wrongly, and in some implementations access may even go outside the array.

20.5.3 Sorting records

In real applications, what gets sorted is usually not a number but a **record** with several fields. In C, records are represented with `struct`. The comparison function determines which field the sort is based on:

```c
typedef struct { char name[20]; int grade; } student;

int by_name(const void *p, const void *q) {
    const student *x = p, *y = q;
    return strcmp(x->name, y->name);        // alphabetical
}
int by_grade_desc(const void *p, const void *q) {
    const student *x = p, *y = q;
    return (y->grade > x->grade) - (y->grade < x->grade);
}
/* qsort(s, n, sizeof s[0], by_name); */
```

The `student` structure carries a 20-character name and a grade; its size is 24 bytes.

In C, a `void *` can be assigned to another object pointer without an explicit cast; `const student *x = p` is therefore valid. `strcmp` already returns a value that meets the contract of `qsort`. In `by_grade_desc`, `x` and `y` are swapped so that the higher grade comes first.

These two functions were tried on the array *{Zeynep 85, Ali 70, Mehmet 85, Ayse 92, Can 70}*. Sorting with `by_name` gives *Ali, Ayse, Can, Mehmet, Zeynep*. Sorting that sorted array afterwards with `by_grade_desc` gave *Ayse 92, Mehmet 85, Zeynep 85, Can 70, Ali 70*. The alphabetical order of the two students with grade 70 was disturbed. This is not a bug. The standard does not require `qsort` to be stable or to use a particular algorithm. If the order among equal keys matters, a secondary key is added to the comparison function:

```c
int by_grade_then_name(const void *p, const void *q) {
    const student *x = p, *y = q;
    int k = (y->grade > x->grade) - (y->grade < x->grade);
    if (k != 0) return k;               /* grades differ: higher grade first */
    return strcmp(x->name, y->name);        /* grades equal: alphabetical */
}
```

With this function the result is *Ayse 92, Mehmet 85, Zeynep 85, Ali 70, Can 70*, independent of the order of the input.

Common mistake

Writing the comparison function as the shortcut `return x - y;`. It works for small values, but when the difference does not fit in an `int`, **signed integer overflow** occurs, which is undefined behavior in C. For *x = INT_MIN, y = 1* the true difference is *−2,147,483,649*, which is below the lower limit of `int`. A program compiled with `-fsanitize=undefined` stopped on this line with the message “signed integer overflow.” In a build where overflow wraps around (`-fwrapv`), the result came out as *+2,147,483,647*, that is, `INT_MIN` was treated as greater than 1. `(x > y) - (x < y)` overflows for no value.

The price of the library routine is that every comparison is a function call. One million random `int`s were sorted in 0.21 seconds with `qsort` and in 0.10 seconds with `hybrid_sort` (`-O2`; times vary by machine). Even so, `qsort` is a tested, ready-made routine that works with every type, and in most programs it is the right choice. Which algorithm it uses varies from implementation to implementation.

20.5.4 Hybrid sorts

No single algorithm is best in every situation. Quicksort is fast on average but degrades to n^2 in the worst case. Heap sort is $n \lg n$ even in the worst case, but its constant is large. Insertion sort is unrivaled on small and nearly sorted arrays. A **hybrid sort** combines these strengths:

- **Quicksort + insertion sort.** On small parts the cost of recursion dominates. Parts smaller than M are finished with insertion sort. `quicksort3` with `hybrid_sort` is an example.
- **Introsort** (Musser, 1997). It starts as quicksort and monitors the recursion depth. If the depth exceeds about $2\lfloor \lg n \rfloor$, that signals that the splits are going badly, and that part is sorted with heap sort. This brings the worst case down to $O(n \lg n)$. Small parts are again finished with insertion sort. It is common in implementations of `std::sort` in the C++ library.
- **Timsort** (Peters, 2002). It combines merge sort with insertion sort. It finds the ready-sorted stretches of the data, the "runs," and merges them. Because it exploits the order present in real data, it approaches linear time on nearly sorted input. It is stable. Python's `sort` and Java's `Arrays.sort` for object arrays are Timsort.

The common idea has three layers: good asymptotic behavior at large scale ($n \lg n$), a low constant at small scale (insertion sort), and an insurance policy for the worst case. This is why the sorting routines in real libraries differ from the pure algorithms in textbooks.

20.6 Merge sort

20.6.1 The idea

Merge sort is a divide-and-conquer algorithm described by John von Neumann in 1945. It can be stated in four sentences:

1. A list of 0 or 1 items is already sorted.
2. Otherwise the list is split into two parts that are as equal as possible.
3. Each part is sorted by applying merge sort recursively.
4. The two sorted sublists are merged into a single sorted list.

The algorithm rests on two observations. First, sorting a small list takes fewer steps. Second, producing one sorted list from two sorted lists is much cheaper than producing one from two unsorted lists. The smallest item of two sorted lists is always at the front; the next item of the merged

list is the smaller of these two fronts. So merging finishes in a number of steps equal to the sum of the lengths of the two lists.

In quicksort every partition puts the pivot in its final place. In merge sort no item settles during the split; the algorithm exploits the advantage of two arrays being sorted. Splitting the input in the middle depended on luck in quicksort; here it is always the case.

20.6.2 `merge`: merging from both ends

The code is the version in Sedgewick's book. Merging in place is hard, so an auxiliary array `b` of the same size as `a` is used.

```
void merge(int a[], int b[], int L, int m, int R) {
    int i, j, k;
    for (i = m + 1; i > L; i--) b[i - 1] = a[i - 1];
    for (j = m; j < R; j++) b[R + m - j] = a[j + 1];
    for (k = L; k <= R; k++)
        if (less(b[j], b[i]))
            a[k] = b[j--];
        else
            a[k] = b[i++];
}
```

`merge(a, b, L, m, R)` merges the halves `a[L..m]` and `a[m+1..R]`, each sorted on its own, into a single sorted array in `a[L..R]`. Line by line:

- **Line 3.** The left half is copied as is into `b[L..m]`. The loop starts at `i = m + 1`, performs the assignment `b[i-1] = a[i-1]`, and ends when `i = L`. After the loop, `i` points to the first item of the left half.
- **Line 4.** The right half is copied into `b[m+1..R]` in *reverse order*. For `j = m`, `b[R] = a[m+1]` is written; for `j = R - 1`, `b[m+1] = a[R]`. The loop ends with `j = R`; `j` points to the smallest item of the right half.
- **Lines 5–9.** `k` is the position to be written in `a`. At each step `b[i]` and `b[j]` are compared and the smaller is written to `a[k]`. If the item comes from the right, `j` decreases; if from the left, `i` increases. On equality `less(b[j], b[i])` is false, and the item on the left is taken.

The reverse copy is the subtlest point of the code. After the copy, `b[L..R]` first rises and then falls: it is a **bitonic** array. In an ordinary merge, every step needs the check “is this half finished?” Here, when the left half is exhausted, `i` enters the large items of the right half, and these are not smaller than `b[j]`; from then on the loop always takes from `j`. The situation for the right half is symmetric: each half is the sentinel of the other.

20.6.3 Code trace: `merge(a, b, 0, 2, 6)`

Let `a = [3, 5, 7, 4, 5, 6, 10]`. The left half is `a[0..2] = 3 5 7` and the right half is `a[3..6] = 4 5 6 10`; both are sorted. After the two copy loops, `b = [3, 5, 7, 10, 6, 5, 4]`, with $i = 0$ and $j = 6$.

`k`	`b[i]`	`b[j]`	**Decision**	`i, j` **(after)**	`a` **(after)**
0	3	4	$4 \geq 3$: `a[0] = b[0]`	1, 6	`3 5 7 4 5 6 10`
1	5	4	$4 < 5$: `a[1] = b[6]`	1, 5	`3 4 7 4 5 6 10`
2	5	5	$5 \geq 5$: `a[2] = b[1]`	2, 5	`3 4 5 4 5 6 10`
3	7	5	$5 < 7$: `a[3] = b[5]`	2, 4	`3 4 5 5 5 6 10`
4	7	6	$6 < 7$: `a[4] = b[4]`	2, 3	`3 4 5 5 6 6 10`
5	7	10	$10 \geq 7$: `a[5] = b[2]`	3, 3	`3 4 5 5 6 7 10`
6	10	10	$10 \geq 10$: `a[6] = b[3]`	4, 3	`3 4 5 5 6 7 10`

Table 20.10: The third loop of `merge(a, b, 0, 2, 6)`. `b = [3, 5, 7, 10, 6, 5, 4]`.

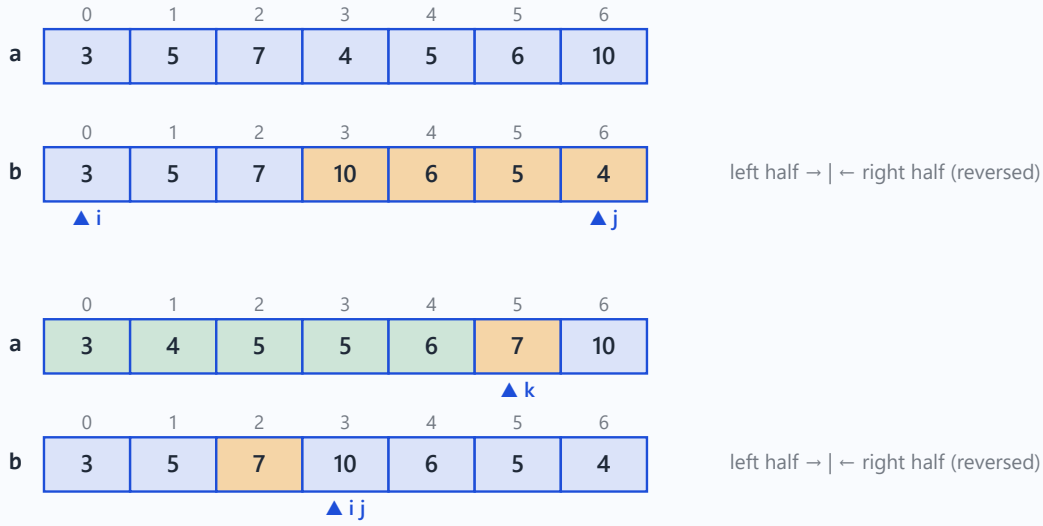


Figure 20.7: Two moments of the `merge` trace. Top: after copying, the left half of `b` is ascending and the right half is reversed; from the ends toward the middle, both sides ascend. Bottom: step $k = 5$, where the last item of the left half, 7, has been taken, and `i` and `j` have met in the same cell.

At $k = 5$ the last item of the left half, 7, is taken and `i` moves to 3, that is, into the right half. At this point `b[i] = b[j] = 10`; both indices point to the same cell. In the last step `i` takes this item. At the end $i = 4$ and $j = 3$; the two indices have met in the middle. No bounds check was made, and none was needed. Among the equal 5s, the one on the left, `b[i]`, was taken first.

Merging copies $R - L + 1$ items and makes exactly one comparison per placement. The time cost is $\Theta(R - L + 1)$; this does not change whatever the sizes of the two halves.

20.6.4 `mergesort` and the auxiliary array

```
void mergesort(int a[], int b[], int L, int R) {
    if (R <= L) return;
    int m = (R + L) / 2;
    mergesort(a, b, L, m);
    mergesort(a, b, m + 1, R);
    merge(a, b, L, m, R);
}
```

The routine returns if the range is empty or has one item. Otherwise it finds the middle with `m = (R + L) / 2`, sorts the two halves, and merges them. If the range has an odd number of items, the left half gets one more: 17 items split as 9 + 8.

The auxiliary array `b` is passed as a parameter. If it were allocated afresh in every `merge` call, each call would perform a `malloc` and a `free`; that means the overhead of $n - 1$ memory requests. The right way is to allocate the array once and hand the same array to all calls. A wrapper function can do this:

```
int merge_sort_alloc(int a[], int n) {
    int *b = malloc(n * sizeof *b);   /* same size as a */
    if (b == NULL) return 0;          /* no memory: not sorted */
    mergesort(a, b, 0, n - 1);
    free(b);
    return 1;
}
```

If the pointer returned by `malloc` is `NULL`, memory could not be allocated. In that case the array is left unsorted and 0 is returned; the caller learns of the situation from this return value. The expression `n * sizeof *b` uses the size of the type that `b` points to, so it stays correct even if the type of `b` changes. When the work is done, the block is returned with `free`. If `free` were forgotten, $4n$ bytes would be lost on every call; this is a **memory leak**. The wrapper was tested with a 10-item array and a counting `malloc` / `free` pair: one allocation, one release.

Common mistake

Not allocating `b[]` at all, that is, declaring `int *b;` and calling `mergesort(a, b, 0, n - 1)` directly. `b` is an uninitialized pointer and points to a random address. `merge` writes to that address; the program either crashes or overwrites other variables. The opposite is also a mistake: allocating `b` with `malloc` inside `merge` on every call. The program runs correctly but slows down needlessly.

20.6.5 Code trace: nine `merge` calls

Let the array be *[89, 67, 59, 62, 23, 93, 24, 20, 28, 60]*. `mergesort(a, b, 0, 9)` first sorts the left half completely. The first merge happens only in the deepest call, *(0, 1)*. The table gives, in order, the nine merges made for ten items.

#	`(L, m, R)`	Halves being merged	`a` (after)
1	(0, 0, 1)	`a[0..0]` + `a[1..1]`	**67 89** 59 62 23 93 24 20 28 60
2	(0, 1, 2)	`a[0..1]` + `a[2..2]`	**59 67 89** 62 23 93 24 20 28 60
3	(3, 3, 4)	`a[3..3]` + `a[4..4]`	59 67 89 **23 62** 93 24 20 28 60
4	(0, 2, 4)	`a[0..2]` + `a[3..4]`	**23 59 62 67 89** 93 24 20 28 60
5	(5, 5, 6)	`a[5..5]` + `a[6..6]`	23 59 62 67 89 **24 93** 20 28 60
6	(5, 6, 7)	`a[5..6]` + `a[7..7]`	23 59 62 67 89 **20 24 93** 28 60
7	(8, 8, 9)	`a[8..8]` + `a[9..9]`	23 59 62 67 89 20 24 93 **28 60**
8	(5, 7, 9)	`a[5..7]` + `a[8..9]`	23 59 62 67 89 **20 24 28 60 93**
9	(0, 4, 9)	`a[0..4]` + `a[5..9]`	**20 23 24 28 59 60 62 67 89 93**

Table 20.11: Trace of `mergesort(a, b, 0, 9)`. Each row is a `merge(L, m, R)` call; the bold part is the range that has just become sorted.

In the 7th merge the array does not change; 28 and 60 are already in order. `merge` does not know this and copies the two items and writes them back anyway. Merge sort does not exploit the order of its input; this is the main result of its analysis.

20.6.6 The frame tree

The structure of the recursive calls can be shown as a tree. Each node is the frame of one `mergesort` call and holds the values *L*, *R*, and *m*. The figure gives this tree. The algorithm descends deep along the left, and merges on the way back. The `merge` at a node happens only after the work of both its children is done; this is a **postorder** traversal of the tree.

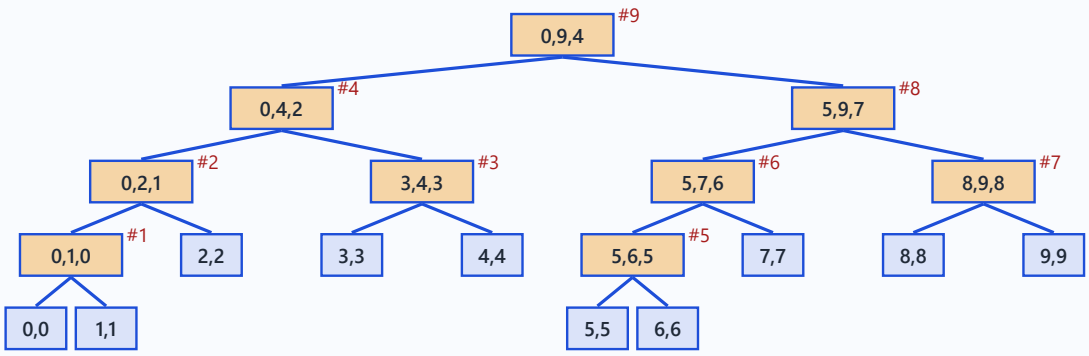


Figure 20.8: The frame tree of `mergesort(a, b, 0, 9)`. The boxes show *L, R, m*. Orange boxes are the frames that call `merge`; the *#k* beside them gives which merge it is. The leaves are single-item ranges and return

immediately.

The tree has 19 nodes: 9 internal nodes perform `merge` and 10 leaves are single-item ranges. The height of the tree is $\lceil \lg 10 \rceil = 4$; at most five `mergesort` frames are on the call stack at one time. The logic of producing one sorted output from two sorted sources comes up again and again in data structures under the name “merge.”

20.6.7 The order of the recursive lines

Suppose the two `mergesort` lines in the routine are swapped, so that the right half is sorted first and then the left half. What changes? The result does not change, because the two halves are independent and `merge` is called after both are done. What changes is the order of the merges and the intermediate tables. The table gives the new order for the same array.

#	Range	Array (after)
1	(8, 9)	`89 67 59 62 23 93 24 20 28 60`
2	(5, 6)	`89 67 59 62 23 24 93 20 28 60`
3	(5, 7)	`89 67 59 62 23 20 24 93 28 60`
4	(5, 9)	`89 67 59 62 23 20 24 28 60 93`
5	(3, 4)	`89 67 59 23 62 20 24 28 60 93`
6	(0, 1)	`67 89 59 23 62 20 24 28 60 93`
7	(0, 2)	`59 67 89 23 62 20 24 28 60 93`
8	(0, 4)	`23 59 62 67 89 20 24 28 60 93`
9	(0, 9)	`20 23 24 28 59 60 62 67 89 93`

Table 20.12: The merges when the two recursive lines are swapped. The right halves merge first.

The frame tree is the same; only the right child is visited first at every node.

20.6.8 Analysis and stability

An array of n items is split into two halves of $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$ items; merging makes n comparisons. The number of comparisons a_n satisfies the recurrence

$$a_n = a_{\lceil n/2 \rceil} + a_{\lfloor n/2 \rfloor} + n,\ a_1 = 0$$

For example, 16 items split as 8 + 8 and 17 items as 9 + 8. If $n = 2^k$, the two halves are equal and the recurrence becomes $a_n = 2a_{n/2} + n$. This is the best-case recurrence of quicksort and gives $a_n = n \lg n$.

Theorem 20.3 (Cost of merge sort)

`mergesort` makes $\Theta(n \lg n)$ comparisons on every array of n items; if $n = 2^k$ the count is exactly $n \lg n$. It makes $2n - 1$ calls and $n - 1$ merges. The extra memory is the n-item auxiliary array plus a call stack of depth $O(\lg n)$.

Proof. Each `merge` makes as many comparisons as there are items in its range, and this number does not depend on the order of the input. So the number of comparisons depends only on n and equals a_n. At every level of the frame tree the ranges cover the array, and the total work of a level is at most n; the number of levels is $\lceil \lg n \rceil$. Hence $a_n \le n\lceil \lg n \rceil$. For the lower bound: every range at depth $d < \lfloor \lg n \rfloor$ has size at least $\lfloor n/2^d \rfloor \ge 2$; each of these first $\lfloor \lg n \rfloor$ levels makes exactly n comparisons, so $a_n \ge n\lfloor \lg n \rfloor$. For the number of calls, the recurrence $T_n = T_{\lceil n/2 \rceil} + T_{\lfloor n/2 \rfloor} + 1$, $T_1 = 1$ gives $T_n = 2n - 1$: for $n \ge 2$, $(2\lceil n/2 \rceil - 1) + (2\lfloor n/2 \rfloor - 1) + 1 = 2n - 1$. The tree has n leaves; the remaining $n - 1$ nodes perform merges.

Instrumented runs confirmed these values. For 10 items: 19 calls, 9 merges, and 34 comparisons; for 16 items: 31 calls, 15 merges, and exactly $16 \cdot 4 = 64$ comparisons; for 17 items: 33 calls and 70 comparisons. The recurrence can also be unrolled by hand: $a_{10} = 2a_5 + 10$, $a_5 = a_3 + a_2 + 5 = 5 + 2 + 5 = 12$, and $a_{10} = 34$.

The running time is independent of the state of the array: the best, average, and worst cases are all $n \lg n$; there is no bad input. The price is $O(n)$ extra memory. Ways of removing it raise the cost; if extra memory is unwelcome, heap sort comes to the fore.

Merging reads and writes sorted pieces sequentially. This property makes it the basis of **external sorting**, that is, sorting files on disk that do not fit in memory: pieces that fit in memory are sorted and written to disk, then merged.

Stability requires a distinction. The classic merge takes the item from the left half on equality and copies the remaining items separately; this merge is **stable**. Quicksort and heap sort are not stable; merge sort differs from them in this respect. However, the bitonic `merge` with reverse copying used here is not stable. When the left half is exhausted, `i` enters the reverse-ordered items of the right half and takes equal items in reverse order. With four equal keys *[2a, 2b, 2c, 2d]*, the sort gave *[2a, 2b, 2d, 2c]*: 2c and 2d swapped places. In an experiment with 1000 random arrays of 2–10 items with values from 0 to 3, the order of equal items was disturbed in 194 arrays. If stability is needed, the right half is not copied in reverse and a check is made for whether one of the halves has run out.

20.7 Linked merge sort

20.7.1 Large records and the `link` field

In real applications an item may be a record of hundreds of bytes, for example one carrying a student's name and address. Copying such records between two arrays at every merge takes time. The solution is not to move the records at all: a `link` field is added to each record, and a linked list is built inside the array using indices. Sorting changes only the indices.

```
typedef struct {
    int key;   /* + large data fields */
    int link;       /* index of the next item, 0 = end */
} element;

int mergesort(element a[], int L, int R) {
    if (R <= L) return L;          /* single-item list */
    int m = (R + L) / 2;
    int right = mergesort(a, m + 1, R);
    int left = mergesort(a, L, m);
    return merge(a, left, right);     /* head index */
}
```

The `element` structure carries a key and a link. The comment "+ large data fields" reminds us that in a real application the structure may be much bigger. `link` is the index in the array of the item that follows this one in the list; 0 marks the end of the list. For this reason the data live in `a[1..n]` and `a[0]` is not used as an item. `a[0]` serves as a temporary list head during merging. Initially all `link` fields are 0: every item is a list by itself.

This time `mergesort` returns a value: the head index of the sorted list built from the items in `a[L..R]`. In a single-item range the list is the item itself and `L` is returned. In a larger range the two halves are sorted separately, the lists they return are merged with `merge`, and the head of the new list is returned.

In the code the right half is sorted first, and this order is deliberately written on two separate lines. If the two calls were written as the arguments of `merge`, it would be unclear which runs first: in C the order of evaluation of arguments is **unspecified**. The result comes out the same, but the intermediate states, and any hand-prepared trace table, would vary with the compiler.

20.7.2 Linked `merge`

```
int merge(element a[], int first, int second) {
    int tail = 0;              // a[0]: temporary head
    int i1 = first, i2 = second;
    while (i1 && i2)
        if (a[i1].key <= a[i2].key) {
            a[tail].link = i1; tail = i1;
            i1 = a[i1].link;
        } else {
            a[tail].link = i2; tail = i2;
            i2 = a[i2].link;
        }
    a[tail].link = i1 ? i1 : i2;
    return a[0].link;
}
```

`merge(a, first, second)` merges the two sorted lists whose heads are `first` and `second` into a single sorted list and returns the head index of the new list. Line by line:

- **Line 2.** `tail` is the last item of the merged list so far. At the start the list is empty and `tail = 0`, that is, the temporary head `a[0]`. This little trick removes the need to write a separate "is it the first item?" case. The first item added is also written through `a[tail].link`, that is, to `a[0].link`.
- **Line 3.** `i1` and `i2` are the first items of the two lists not yet taken.
- **Lines 4–11.** The loop continues while both lists have items left. The item with the smaller key is linked to the end of the merged list (`a[tail].link = i1`), `tail` advances to it, and the index of the list it came from moves to the next item. On equality, thanks to `<=`, the item in the first list is taken. As long as the first list comes from the left half, equal keys keep their input order; this version is stable.
- **Line 12.** When one list runs out, the rest of the other is attached with a single link. As in the array version, there is no need to copy the remaining items one by one; the remainder is already a sorted list.
- **Line 13.** The head of the merged list is in `a[0].link` and is returned.

No item moves; only links are rewritten. The cost of a merge is the number of loop iterations, which is less than the sum of the lengths of the two lists. The number of comparisons comes out slightly smaller than in the array version, because comparison also stops when one list ends.

20.7.3 Code trace: the last merge

Let the keys be `a[1..10] = 71, 77, 122, 38, 119, 41, 22, 115, 43, 13`. The last, ninth merge of `mergesort(a, 1, 10)` merges two lists of five items each. The first list (head 4) is *38 → 71 → 77 → 119 → 122* and the second list (head 10) is *13 → 22 → 41 → 43 → 115*.

Round	Comparison	Link written	`tail`	`i1`	`i2`
0	—	—	0	4	10
1	*38 > 13*	`a[0].link = 10`	10	4	7
2	*38 > 22*	`a[10].link = 7`	7	4	6
3	*38 ≤ 41*	`a[7].link = 4`	4	1	6
4	*71 > 41*	`a[4].link = 6`	6	1	9
5	*71 > 43*	`a[6].link = 9`	9	1	8
6	*71 ≤ 115*	`a[9].link = 1`	1	2	8
7	*77 ≤ 115*	`a[1].link = 2`	2	5	8
8	*119 > 115*	`a[2].link = 8`	8	5	0
end	`i2 = 0`: the second list is exhausted	`a[8].link = 5`	8	5	0

Table 20.13: The last call of the linked `merge`, `merge(a, 4, 10)`.

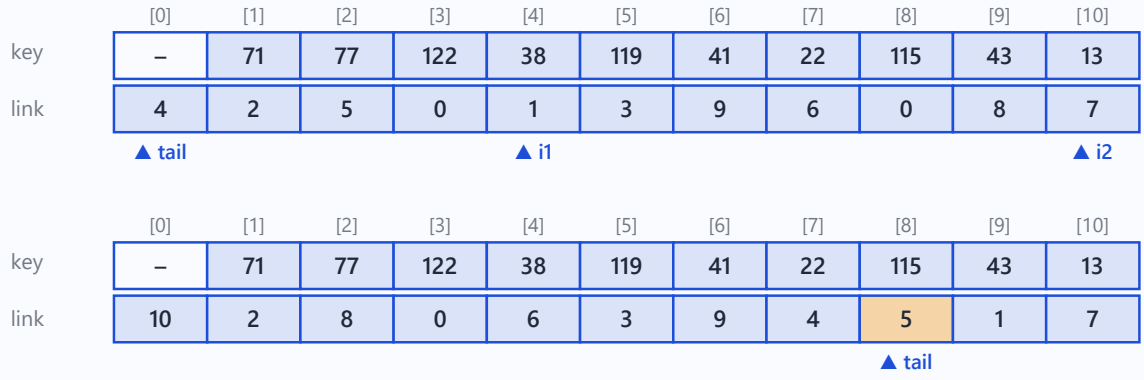


Figure 20.9: The start (top) and end (bottom) of the last merge. The top row shows the keys and the bottom row the `link` fields. The items never move; only the links change.

The function returns `a[0].link = 10`. Following the links reads off the sorted list: *13 → 22 → 38 → 41 → 43 → 71 → 77 → 115 → 119 → 122*. Eight comparisons were made in eight rounds, and the remaining two items (119 and 122) were attached with a single link.

20.7.4 How the `link` fields change

The table gives all nine merges over the whole sort. Since the right half is sorted first in the code, the first merge is on the rightmost range *(9, 10)*.

#	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	Return	*L, R*
key	71	77	122	38	119	41	22	115	43	13		
1	0	0	0	0	0	0	0	0	0	**9**	10	9, 10
2	0	0	0	0	0	0	**6**	0	0	9	7	6, 7
3	0	0	0	0	0	**8**	6	0	0	9	7	6, 8

#	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	Return	*L, R*
4	0	0	0	0	0	**9**	6	0	**8**	**7**	10	6, 10
5	0	0	0	**5**	0	9	6	0	8	7	4	4, 5
6	**2**	0	0	5	0	9	6	0	8	7	1	1, 2
7	2	**3**	0	5	0	9	6	0	8	7	1	1, 3
8	2	**5**	0	**1**	**3**	9	6	0	8	7	4	1, 5
9	2	**8**	0	**6**	3	9	**4**	**5**	**1**	7	10	1, 10

Table 20.14: The `link` fields after each `merge` in linked merge sort. Bold: the link changed in that merge.

Each row is read by starting from the return value and following the links. In row 4 the return value is 10: *13 → 22 → 41 → 43 → 115*; the right half *a[6..10]* is sorted. In row 8 the return value is 4: *38 → 71 → 77 → 119 → 122*.

At the end the items are still in their old places in the array; the sorted output is written by walking the list (`for (p = head; p; p = a[p].link)`). The extra memory is one `int` per item; the number of comparisons is *O(n lg n)* and the number of record moves is zero.

Common mistake

Three mistakes are common when writing the linked version. The first is declaring `mergesort` as `void` out of habit and then trying to return a value at the end; the compiler reports this as an error. The second is writing `i1 = a[i1].link` instead of `i2 = a[i2].link` in the `else` branch. Then the second list never advances, the first list advances in the wrong place, and the lists are corrupted. The third is returning `tail` instead of `a[0].link` ; `tail` points to the end of the list, not its head.

20.7.5 An exam question

The question below uses another way of writing the linked version. In the question the items are `a[1..9]` and are sorted by their keys. The tasks are: (i) draw the stack frames of the recursive calls and write the return value if `merge` was called; (ii) fill in the execution table with the `merge` numbers, the return values, and the `link` fields at each step.

```
int mergesort(element a[], int L, int R) {
    int m;
    if (R <= L) return L;
    m = (R + L) / 2;
    return merge(a, mergesort(a, m + 1, R),
                    mergesort(a, L, m));
}
```

The keys are `[1] ?, [2] ?, [3] 18, [4] -5, [5] 27, [6] 11, [7] ?, [8] 83, [9] 30`. In place of the question marks on the exam paper, 42, 7, and 55 are used here. `merge` is the same as in linked merge sort. In the printed version of the question, `i1` was written in the `else` branch; this is a typo and the correct one is `i2`.

The first trap of the question is that the recursive calls are the arguments of `merge`; which half is sorted first is left to the compiler. The solution assumes that the arguments are evaluated from left to right, that is, the right half is sorted first. The call order is: *(1, 9) → (6, 9) → (8, 9) → (9, 9) → (8, 8) → (6, 7) → (7, 7) → (6, 6) → (1, 5) → …*. The second trap is that the right half is passed to `merge` as the first list. Since the first list is taken first on equality, this way of writing is not stable; because the keys in this question are distinct, it does not affect the result.

#	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	Return	*L, R, m*
1	0	0	0	0	0	0	0	0	**8**	9	8, 9, 8
2	0	0	0	0	0	**7**	0	0	8	6	6, 7, 6
3	0	0	0	0	0	**9**	**8**	0	**7**	6	6, 9, 7
4	0	0	0	**5**	0	9	8	0	7	4	4, 5, 4
5	0	**1**	0	5	0	9	8	0	7	2	1, 2, 1
6	0	**3**	**1**	5	0	9	8	0	7	2	1, 3, 2
7	0	3	**5**	**2**	**1**	9	8	0	7	4	1, 5, 3
8	**7**	**6**	5	2	**9**	**3**	8	0	**1**	4	1, 9, 5

Table 20.15: Solution of the exam question (arguments left to right). Keys: 42, 7, 18, −5, 27, 11, 55, 83, 30.

The head of the result list is 4: *−5 → 7 → 11 → 18 → 27 → 30 → 42 → 55 → 83*. The table was produced by a C program in which the evaluation order is written explicitly. If the arguments were evaluated from right to left, the first merge would be at *(1, 2)* and the intermediate rows would change; the last row and the result would stay the same.

20.8 Heap sort

Merge sort is *O(n lg n)* in every case but needs *O(n)* extra memory. Quicksort works in place but degrades to n^2 in the worst case. **Heap sort** combines two good properties: it runs in *O(n lg n)* even in the worst case and uses no extra memory. It does this by building the heap structure, used to implement priority queues, inside the array being sorted.

20.8.1 An array is a tree

The array is viewed as `a[1..n]`; `a[0]` is not used. This array is read like a **complete binary tree**: the root is `a[1]`, the children of `a[i]` are `a[2i]` and `a[2i+1]`, and its parent is `a[i/2]`. The nodes of the tree are placed in the array level by level, left to right. So no pointers are needed; the

links come from index arithmetic. One-based indexing simplifies these formulas. In a zero-based array the children would be *2i + 1* and *2i + 2*.

Definition 20.3

A complete binary tree in which the value of every node is not less than the values of its children is called a **max-heap**. In the array representation this condition means *a[i/2] ≥ a[i]* for every *2 ≤ i ≤ n*. In a max-heap the largest item is at the root.

Heap sort works in two phases. First, the array is turned into a max-heap. Then the largest item at the root is moved to the end of the array, the heap shrinks by one item, and the remaining heap is repaired. When this step is repeated *n − 1* times, the array is sorted from smallest to largest.

```
void heapify(int a[], int root, int n) {
    int t = a[root], j;
    for (j = 2 * root; j <= n; j *= 2) {
        if (j < n && a[j] < a[j + 1]) j++; // larger
        if (t >= a[j]) break;   // t fits here
        a[j / 2] = a[j];        // move the child up
    }
    a[j / 2] = t;
}
void heapsort(int a[], int n) {     /* a[1..n] */
    int i;
    for (i = n / 2; i >= 1; i--)     /* build the heap */
        heapify(a, i, n);
    for (i = n - 1; i >= 1; i--) {   /* sort */
        SWAP(a[1], a[i + 1]);    /* largest to the end */
        heapify(a, 1, i);            /* repair the heap */
    }
}
```

20.8.2 `heapify`

`heapify(a, root, n)` finds the proper place for the value at the root, in the heap `a[1..n]`, when the left and right subtrees of node `root` are already max-heaps. This is exactly the **sift-down** operation performed when removing the largest item from a priority queue; it appears in Horowitz's book under the name `adjust`. Line by line:

- **Line 2.** The value at the root is taken into `t`. The root now holds a “hole.” `t` will finally be written wherever this hole ends up.
- **Line 3.** `j` is the left child of the hole: `2 * root`. At each round the loop descends one level (`j *= 2`) and ends when `j` leaves the heap.

- **Line 4.** If the right child exists (`j < n`) and is larger than the left child, `j` moves to the right child. `j` now points to the larger child.
- **Line 5.** If `t` is not smaller than the larger child, the hole is exactly in place and the loop exits.
- **Line 6.** Otherwise the larger child is moved up into the hole, that is, to its parent `a[j/2]` . The hole descends to `j` .
- **Line 8.** Wherever the loop ends, the hole is at `a[j/2]` and `t` is written there. When the exit is by `break` , `j/2` is the hole itself. When the loop ends with `j > n` , `j` would be a child of the last hole; `j/2` again gives the hole.

Shifting is done instead of swapping: one assignment per level and one final assignment for `t` . The cost of `heapify` is proportional to the number of levels the hole descends and cannot exceed *lg n*.

20.8.3 Building the heap

The first way that comes to mind for turning an array into a heap is to insert the items one by one into an empty heap. In applications such as Kruskal's algorithm, the items arrive one at a time anyway, so the heap is built this way. Since each insertion takes *O(lg n)*, the total is *O(n lg n)*. If the items are already available in an array, there is a better way: **bottom-up heap construction**.

The leaves are already one-item heaps. Nodes with $i > \lfloor n/2 \rfloor$ have no children, because their left children would fall at position $2i > n$. So the loop starts at $\lfloor n/2 \rfloor$ and goes backward toward the root: $\lfloor n/2 \rfloor, \lfloor n/2 \rfloor - 1, \ldots, 1$. For each i the subtrees of the node have already been repaired, so the precondition of `heapify` holds. When the loop reaches the root, the whole array is a max-heap.

Horowitz's example is the array *[26, 5, 77, 1, 61, 11, 59, 15, 48, 19]*; $n = 10$, $\lfloor n/2 \rfloor = 5$.

Call	`t`	Event	Shift	`a[1..10]` (after)
`heapify(a, 5, 10)`	61	single child 19; *61 ≥ 19*	0	`26 5 77 1 61 11 59 15 48 19`
`heapify(a, 4, 10)`	1	children 15, 48; 48 moves up, 1 goes to `a[9]`	1	`26 5 77 48 61 11 59 15 1 19`
`heapify(a, 3, 10)`	77	children 11, 59; *77 ≥ 59*	0	same
`heapify(a, 2, 10)`	5	61 moves up; child 19 moves up; 5 goes to `a[10]`	2	`26 61 77 48 19 11 59 15 1 5`
`heapify(a, 1, 10)`	26	77 moves up; 59 moves up; 26 goes to `a[7]`	2	`77 61 59 48 19 11 26 15 1 5`

Table 20.16: The heap-building phase. A shift is the moving of a child up into its parent (`a[j / 2] = a[j]`).

The building phase made 5 shifts in total. The result is a valid max-heap: the root is 77 and every node is larger than its children. For example, the children of `a[2] = 61` are `a[4] = 48` and `a[5] = 19` .

20.8.4 The sorting phase

Once the array is a heap, the largest item is at the root. This item is swapped with the last cell of the array: `SWAP(a[1], a[i + 1])`. The largest item is now in its final place. The heap shrinks by one item and becomes `a[1..i]`. The item that arrives at the root will most likely break the heap condition, but both subtrees are still heaps; `heapify(a, 1, i)` repairs it. The sorted region accumulating at the end of the array grows by one cell per round.

`i`	Sent to the end	Item brought to the root, where it lands	`a[1..10]` (after)
9	77	5 → `a[8]`	`61 48 59 15 19 11 26 5 1 | 77`
8	61	1 → `a[7]`	`59 48 26 15 19 11 1 5 | 61 77`
7	59	5 → `a[5]`	`48 19 26 15 5 11 1 | 59 61 77`
6	48	1 → `a[6]`	`26 19 11 15 5 1 | 48 59 61 77`
5	26	1 → `a[4]`	`19 15 11 1 5 | 26 48 59 61 77`
4	19	5 → `a[2]`	`15 5 11 1 | 19 26 48 59 61 77`
3	15	1 → `a[3]`	`11 5 1 | 15 19 26 48 59 61 77`
2	11	1 → `a[2]`	`5 1 | 11 15 19 26 48 59 61 77`
1	5	1 stays in place	`1 | 5 11 15 19 26 48 59 61 77`

Table 20.17: The sorting phase. Each row is a `SWAP(a[1], a[i + 1])` followed by `heapify(a, 1, i)`; the right of the vertical line is the sorted region.

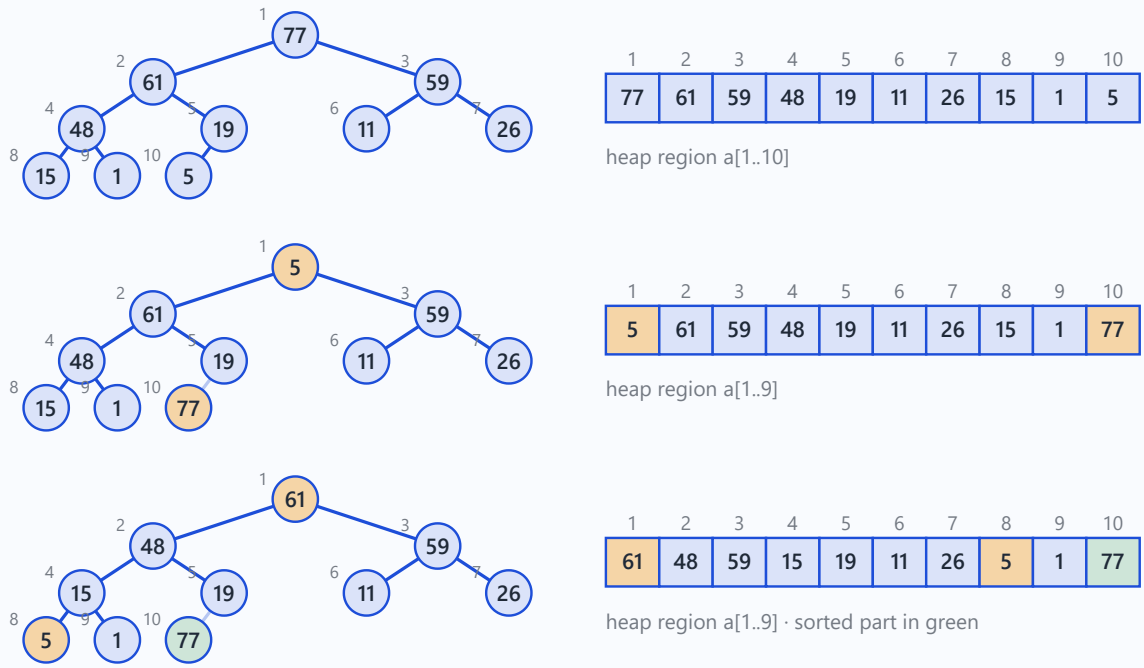


Figure 20.10: The first round of the sorting phase, top to bottom: the built heap; after `SWAP(a[1], a[10])`, 77 has gone to the end and 5 at the root breaks the heap; after `heapify(a, 1, 9)` the new root is 61. In each figure the tree is on the left and the same array on the right. Orange items are those exchanged in that step; green items are the sorted region.

In the last round $i = 1$. 5 goes to the end and a single item, 1, remains at the root. A one-item tree is already a heap. In `heapify(a, 1, 1)`, since $j = 2 > 1$, the loop is never entered and 1 stays where it is. The array is now sorted: *[1, 5, 11, 15, 19, 26, 48, 59, 61, 77]*. The code was also tested on 2000 random arrays of 1 to 200 items with repeated keys and sorted all of them correctly.

To the question "We understood the heap, but when does the array get sorted?" the answer is that a heap is an intermediate order: it knows only the position of the largest item. The sort uses this knowledge $n - 1$ times.

Common mistake

Mixing one-based indices with a zero-based array. The formulas in which the children are $2i$ and $2i + 1$ are correct only for `a[1..n]`. If a C array is declared as `int a[10]` and `heapsort(a, 10)` is called, the routine accesses `a[10]`, outside the array, and the item in `a[0]` is never sorted. The correct approach is to declare the array with `n + 1` items or to rewrite the formulas in their zero-based form.

20.8.5 Analysis

By a crude analysis, `heapify` makes at most $\lg n$ shifts and is called $n/2$ times: $O(n \lg n)$. This bound is loose, because most calls happen in small subtrees.

For a finer analysis the tree is examined level by level. Let k be the number of levels of the tree; $k \approx \lg(n + 1)$. Level i has at most 2^{i-1} nodes. A node on this level can sink at most $k - i$ levels. In the worst case the total number of shifts is

$$\Sigma_{i=1}^{k} 2^{i-1}(k - i) = k \Sigma_{i=1}^{k} 2^{i-1} - \Sigma_{i=1}^{k} i \cdot 2^{i-1}$$

The first sum is a geometric series: $2^k - 1$. The second sum is found from the derivative of the geometric series. Let $S(y) = 1 + y + \ldots + y^k = (y^{k+1} - 1)/(y - 1)$. Differentiating term by term gives $S'(y) = \Sigma_{i=1}^{k} i \cdot y^{i-1}$; the sum we want is $S'(2)$. Differentiating the fraction and setting $y = 2$ gives $S'(2) = (k + 1)2^k - (2^{k+1} - 1) = (k - 1)2^k + 1$. Thus

$$k(2^k - 1) - (k - 1)2^k - 1 = 2^k - k - 1$$

Theorem 20.4 (Cost of heap sort)

In a tree with k levels, bottom-up heap construction makes at most $2^k - k - 1 \le 2n$ shifts; heap construction is $O(n)$. The sorting phase takes $O(n \lg n)$ with $n - 1$ `heapify` calls. Heap sort is $O(n \lg n)$ even in the worst case, uses $O(1)$ extra memory, and is not stable.

Proof. The construction bound is the computation above. A complete binary tree with k levels has at least 2^{k-1} nodes; so $n \geq 2^{k-1}$ and $2^k \leq 2n$. In the sorting phase `heapify(a, 1, i)` makes at most $\lfloor \lg i \rfloor$ shifts; $\Sigma_{i=1}^{n-1} \lg i \leq n \lg n$. The routine uses only a few variables and is non-recursive. One example suffices for instability: the array *[2a, 2b, 1]* is sorted as *[1, 2b, 2a]*.

That the bound is tight can be shown with the input that produces the worst case. In an ascending sorted array, the value of every internal node is smaller than its entire subtree, and every `heapify` sinks the hole all the way to a leaf. For ascending arrays of $n = 2^k - 1$ items, the counted number of shifts matched the formula exactly (the table).

k	$\Sigma\, 2^{i-1}(k-i)$	$2^k - k - 1$ (measured)	$n = 2^k - 1$
2	1	1	3
3	4	4	7
4	11	11	15
6	57	57	63
10	1013	1013	1023
20	1,048,555	1,048,555	1,048,575

Table 20.18: The worst case of heap construction. The two middle columns are identical; the last column is the number of nodes. The number of shifts is always less than n.

The bound "$\leq 2n$" means $O(n)$, not $o(n)$: the number of shifts grows at the same rate as n.

The number of comparisons was also measured. Each shifting level involves two comparisons: one between the two children and one between `t` and the larger child. On random arrays the total came out at 16,899 comparisons for 1000 items, 1.41 million for 50,000 items, and 36.8 million for one million items. These are about 85–92% of $2n \lg n$. The share of the construction phase is small: 1.88 million comparisons for one million items, that is, fewer than $2n$.

20.8.6 Advanced heap sort and knowing your data

About $2n \lg n$ comparisons is more than quicksort's $1.39\, n \lg n$, and heap sort accesses memory irregularly. It is a reasonable choice for data up to around fifty thousand items; for larger data there are versions that reduce the number of comparisons.

The best known of these is **bottom-up heapsort**. Its idea is this: the item `t` that replaces the one taken from the root almost always sinks all the way to a leaf, because it came from the end of the array and is small. So comparing `t` with the larger child at every level is mostly wasted. In the bottom-up version, the hole is sent down to a leaf along the larger children without comparing against `t`. Then `t` is placed, starting at the leaf and moving upward. This walk upward usually ends in a step or two. As a result, one comparison per level is made most of the time instead of two. Versions that are even better in theory are known.

If some properties of the data to be sorted are known, the algorithm is chosen according to them. In counting sort the keys are known to lie in a small range, and sorting takes linear time with no comparisons at all. There are also applications that use Shell sort and bucket sort together. A few basic methods of this kind should be known, and the structure of the data should be recognized. Radix sort is another example of this way of thinking.

20.9 Radix sort

20.9.1 Sorting on multiple keys

Some sorts are done on more than one key. A list of people can be sorted first by last name, with people sharing a last name sorted by first name. Let the keys be denoted $K_0, K_1, \ldots, K_{r-1}$. K_0 is the **most significant** key (most significant digit, MSD) and K_{r-1} is the **least significant** key (least significant digit, LSD). In the list-of-people example, the last name is the most significant key. A three-digit number can be thought of in the same way: the hundreds digit is K_0, the tens digit K_1, and the units digit K_2.

In the MSD approach, items are first divided into groups by the most significant key and each group is sorted internally; this approach is recursive and complicated. The LSD approach looks backward at first glance but is much simpler.

Definition 20.4

Radix sort, in its LSD form, sorts the items first by the least significant digit, then by successively more significant digits, and finally by the most significant digit. Each pass must be **stable**: items with equal digits must keep the order left by the previous pass.

The reason for the stability requirement is this. In the hundreds pass, 208 and 271 have the same hundreds digit, 2. This pass cannot tell them apart. The correct order was determined by the earlier passes: the tens pass put 208 (tens digit 0) before 271 (tens digit 7). If the hundreds pass does not disturb this order, the result is correct. Each pass preserves the work of the previous one and adds a more significant digit on top.

Radix sort never compares the numbers with each other. It is therefore a **non-comparison** sort and is not bound by the $n \lg n$ lower bound of comparison sorts. The price is that the structure of the keys, that is, their digits, must be known.

20.9.2 Data structure and building the list

```
#define MAX_DIGIT 3
#define RADIX 10
typedef struct item *item_ptr;
struct item {
    int digit[MAX_DIGIT];  /* 179 → {1, 7, 9} */
    item_ptr link;
};
```

Each item is a node of a linked list. The `digit` array holds the digits of the number: for 179, *{1, 7, 9}*. `digit[0]` is the hundreds digit (MSD), `digit[1]` the tens, and `digit[2]` the units digit (LSD). The numbers are in the range *[0, 999]*. `link` is the address of the next node in the list.

`typedef struct item *item_ptr;` comes before the structure definition and names the pointer type that points to `struct item`. Because the size of a pointer is independent of the contents of the structure, the structure may not yet be defined; this allows `item_ptr link;` to be written inside the structure. A `typedef` that gives no name, such as `typedef struct item {…};`, is meaningless. On x86-64, `sizeof(struct item)` is 24 bytes: a 12-byte array, 4 bytes of alignment padding, and an 8-byte pointer.

Building the list to be sorted also requires memory management. The following two routines were added to this note. `build_list` builds a list in the same order from an `int` array:

```
/* builds a linked list from 3-digit numbers; frees what was built if memory runs out */
item_ptr build_list(const int x[], int n) {
    item_ptr head = NULL, last = NULL;
    for (int k = 0; k < n; k++) {
        item_ptr d = malloc(sizeof *d);
        if (d == NULL) {              /* stopped midway: avoid a leak */
            free_list(&head);
            return NULL;
        }
        d->digit[0] = x[k] / 100;     /* hundreds */
        d->digit[1] = x[k] / 10 % 10; /* tens */
        d->digit[2] = x[k] % 10;      /* units */
        d->link = NULL;
        if (last) last->link = d; else head = d;
        last = d;
    }
    return head;
}
```

```
/* frees every node of the list and sets the caller's pointer to NULL */
void free_list(item_ptr *pp) {
    while (*pp) {
        item_ptr next = (*pp)->link;
        free(*pp);
        *pp = next;
    }
}
```

`build_list` allocates one node per number. If `malloc` returns `NULL`, memory has run out. If the half-built list were not released, the nodes allocated up to that point would leak. So the routine first deletes the part it has built with `free_list`, then returns `NULL`. The digits are separated with integer division and remainder: *179 / 100 = 1*, *179 / 10 % 10 = 7*, *179 % 10 = 9*. The pointer `last` holds the last node of the list; a new node is linked behind it, so the list keeps the input order.

`free_list` takes its parameter as `item_ptr *pp`, that is, a **pointer to a pointer**. This way it not only releases the nodes but also sets the caller's head pointer to `NULL`. At each round the loop first saves the address of the next node in `next`, then releases the node. If the order were reversed, that is, if `free(*pp)` came first and `(*pp)->link` were read afterwards, freed memory would be accessed. That is undefined behavior. The two routines were tested with a counting `malloc` / `free` pair: 500 random lists were built, sorted, and deleted, and 74,115 allocations were matched by 74,115 releases. In an experiment where memory ran out at the fourth node, the first three nodes were released and `NULL` was returned.

20.9.3 `radixsort`

Written as a function, with the declaration of the bucket arrays, the loop body from the slides takes this form:

```
item_ptr radixsort(item_ptr p) {
    item_ptr front[RADIX], back[RADIX];  /* bucket heads and tails */
    for (int i = MAX_DIGIT - 1; i >= 0; i--) {
        for (int j = 0; j < RADIX; j++)
            front[j] = back[j] = NULL;
        for (; p; p = p->link) {     // distribute
            int b = p->digit[i];
            if (!front[b]) front[b] = p;
            else back[b]->link = p;
            back[b] = p;
        }
        for (int j = RADIX - 1; j >= 0; j--) // collect
            if (front[j]) { back[j]->link = p; p = front[j]; }
    }
    return p;
}
```

For each digit value there is a **bucket** in the form of a linked list. `front[j]` points to the first node of the bucket and `back[j]` to the last; as in a queue, keeping the last node allows appending at the back in constant time.

- **Outer loop.** `i` goes down from `MAX_DIGIT - 1` to 0: first the units digit (*i = 2*), then the tens, and finally the hundreds digit is processed. This is LSD order.
- **Emptying the buckets.** At the start of every pass, the front and back pointers of the ten buckets are set to `NULL`.
- **Distributing.** The list is walked from start to end. The *i*-th digit of each node is taken into `b`. If the bucket is empty, the node becomes the first node of the bucket. Otherwise the `link` of the bucket's last node is linked to this node. In both cases the node is the new end of the bucket. Because nodes are appended to the bucket in order of arrival, distribution is stable. The loop advances with `p = p->link`. This read uses the node's old `link`; the node's own link has not been changed yet, only the link of the previous node in the bucket. When the loop finishes, `p` is `NULL`.
- **Collecting.** The buckets are walked from 9 down to 0. The last node of each non-empty bucket is linked to the head of the list built so far (`back[j]->link = p`), and the head of the list becomes the first node of the bucket (`p = front[j]`). The list is built from back to front; bucket 0, added last, stands at the very front. So the bucket with the small digit comes first. The `link` of the last node of the first bucket added is linked to `p`, which is `NULL` at that moment; the end of the list is closed correctly.
- **Return.** After three passes the list is sorted and its head pointer is returned.

20.9.4 Code trace

Let the list be *179, 208, 306, 093, 859, 984, 055, 009, 271, 033*. The numbers are written with three digits because the structure holds three digits for each number. The first pass is on the units digit. The table gives the distribution step by step.

Item	`digit[2]`	Bucket state (after)
179	9	9: 179
208	8	8: 208 · 9: 179
306	6	6: 306 · 8: 208 · 9: 179
093	3	3: 093 · 6: 306 · 8: 208 · 9: 179
859	9	… · 9: 179 859
984	4	… · 4: 984 · …
055	5	… · 5: 055 · …
009	9	… · 9: 179 859 009
271	1	1: 271 · …

Item	`digit[2]`	Bucket state (after)
033	3	1: 271 · 3: 093 033 · 4: 984 · 5: 055 · 6: 306 · 8: 208 · 9: 179 859 009

Table 20.19: Distribution in the units pass (*i = 2*). Each item is appended to the back of its own bucket.

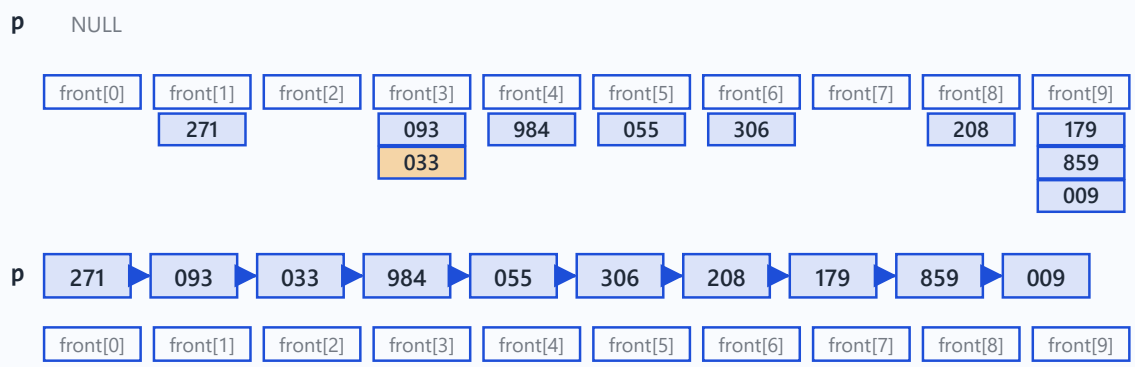


Figure 20.11: The units pass. Top: distribution is finished, and each bucket carries its items in order of arrival. Bottom: the list after collecting: *271 093 033 984 055 306 208 179 859 009*.

In bucket 9, 179, 859, and 009 stand in this order; all three arrived in this order in the input. Collecting walks the buckets from 9 to 0 and links each bucket in front of the list. As a result the small buckets are in front: *271 093 033 984 055 306 208 179 859 009*. The list is now sorted by the units digit.

The second and third passes are done in the same way on the tens and hundreds digits. The table gives the list after each pass.

Pass	List
start	`179 208 306 093 859 984 055 009 271 033`
i = 2 (units)	`271 093 033 984 055 306 208 179 859 009`
i = 1 (tens)	`306 208 009 033 055 859 271 179 984 093`
i = 0 (hundreds)	`009 033 055 093 179 208 271 306 859 984`

Table 20.20: The list after each pass.





Figure 20.12: The hundreds pass. Top: distribution, with four items (009, 033, 055, 093) in bucket 0, in the order left by the tens pass. Bottom: the sorted list after collecting.

After the tens pass, bucket 0 holds 306, 208, and 009; all their tens digits are 0 and they are arranged in the order left by the units pass. In the hundreds pass, 009, 033, 055, and 093 fall into bucket 0. Thanks to stability, these four items keep their order from the tens pass and the bucket comes out sorted on its own. After three passes the list is sorted. The routine was also tested on 500 random lists of 1 to 300 items.

20.9.5 Analysis

Each pass does three jobs: `RADIX` buckets are emptied, n nodes are distributed, and `RADIX` buckets are collected. The number of passes is `MAX_DIGIT`. The total time is

$$O(MAX_DIGIT \cdot (RADIX + n + RADIX)) = O(MAX_DIGIT \cdot (RADIX + n))$$

If the number of digits is denoted d and the radix r, this is written $O(d(n + r))$. In the example it is $O(3 \cdot (10 + n))$ and looks linear. But this is a special case, as in counting sort: the number of digits and the radix are taken to be constants. To tell n distinct numbers apart, at least $\log_r n$ digits are needed. If the range of the keys grows with n, d grows too, and the time again approaches order $n \lg n$.

For the algorithm to be usable, the data must be (a) non-negative, (b) integers, and (c) in a small range. If the range grows, the algorithm still sorts, but the number of passes and the complexity rise quickly. That it can sort only integers is a real disadvantage. It can be adapted to floating-point numbers, but by the time that adaptation is done, the other algorithms would long since have finished the job.

The extra memory is $2 \cdot RADIX$ pointers: the front and back pointers of the buckets. The list is relinked in place. The `link` pointer in every node is also a memory cost, but this cost comes from the linked-list representation, not from the algorithm. Radix sort is stable and makes no comparisons.

20.9.6 Exam question: a modified radix sort

A student thinks that the routine should handle the most significant digit first. The student changes the outer loop to `for (i = 0; i <= MAX_DIGIT-1; i++)` and the collecting loop to `for (j = 0; j <= RADIX-1; j++)`. The question asks you to run the routine on the list *346, 180, 008, 930, 023, 451, 670, 098, 203, 007*, to write the list at each step, and to interpret the result.

Pass	List
start	`346 180 008 930 023 451 670 098 203 007`
i = 0 (hundreds)	`930 670 451 346 203 180 008 023 098 007`
i = 1 (tens)	`098 180 670 451 346 930 023 203 008 007`
i = 2 (units)	`098 008 007 346 023 203 451 180 670 930`

Table 20.21: The passes of the modified routine.

The effect of the two changes is interpreted separately. The first change reverses the order of the passes. Since the last pass is on the units digit, the units digit acts as the most significant digit. If the numbers are read backward, the list is sorted: 098, 008, 007 read backward are 890, 800, 700. The second change reverses the collecting loop. Since the buckets are walked from 0 to 9 and each is linked in front of the list, bucket 9, added last, goes to the front. The order is from largest to smallest, not smallest to largest. As a result, the list is in descending order by the numbers read backward: 890, 800, 700, 643, 320, 302, 154, 081, 076, 039. The table was produced by a C program written with the modified loops.

To sort by MSD, each group formed by the most significant digit must be sorted separately; keeping the groups together and moving on to the next digit spoils the work of the previous pass.

20.10 Comparison and choosing an algorithm

The table compares the elementary and advanced sorting algorithms; the time columns give the order of the number of steps.

Algorithm	Best	Average	Worst	Extra memory	Stable	In place
Selection	n^2	n^2	n^2	$O(1)$	no	yes
Insertion	n	n^2	n^2	$O(1)$	yes	yes
Bubble (early exit)	n	n^2	n^2	$O(1)$	yes	yes
Shell (Knuth gaps)	$n \lg n$	—	$n^{3/2}$	$O(1)$	no	yes
Quick	$n \lg n$	$1.39\, n \lg n$	n^2	$O(\lg n)$ stack*	no	yes
Merge	$n \lg n$	$n \lg n$	$n \lg n$	$O(n)$	yes**	no
Heap sort	$n \lg n$	$n \lg n$	$n \lg n$	$O(1)$	no	yes
Counting	$n + k$	$n + k$	$n + k$	$O(n + k)$	yes	no
Radix (d digits, radix r)	$d(n + r)$	$d(n + r)$	$d(n + r)$	$O(r)$***	yes	—

Table 20.22: Comparison of sorting algorithms. * If the smaller part is processed first; otherwise $O(n)$ in the worst case. ** With the classic merge; the bitonic `merge` in this note is not stable. *** On a linked list.

The lower bound for comparison sorting is $\lceil \lg n! \rceil \approx n \lg n - 1.44\, n$. Among the comparison-based algorithms in the table, the average case of quicksort, merge sort, and heap sort approach this bound within a constant factor. Counting sort and radix sort make no comparisons and so are not bound by

it; in exchange, they need additional information about the keys.

Even where the theoretical orders are the same, real times differ. The table gives the times of this note's routines on random `int` arrays.

n	`quicksort1`	`hybrid_sort`	`quicksort2`	`qsort`	`mergesort`	`heapsort`
100,000	0.008	0.008	0.008	0.017	0.006	0.010
1,000,000	0.130	0.101	0.130	0.211	0.076	0.180

Table 20.23: Time on random `int` arrays, in seconds (`zig cc -O2`, x86-64; average of three measurements). Times vary by machine; only the ratios are meaningful.

In this measurement merge sort came out fastest. The inner loop of the bitonic `merge` has no bounds check and memory is always read and written sequentially; this pattern suits the cache of modern processors very well. The price is a 4 MB auxiliary array for one million items. Heap sort is among the slowest, because the paths from root to leaf make large jumps in memory. The slowness of `qsort` comes from the function call in every comparison. `hybrid_sort`, with median-3 and a cutoff, is about 20% faster than `quicksort1`. These results may vary with the compiler, the processor, and the data.

Which algorithm to choose in which situation can be summarized as follows:

- **General-purpose, in-memory sorting.** Quicksort with median-3 and insertion sort on small parts, or the library `qsort`.
- **Worst-case guarantee.** Heap sort, which works in place, or introsort.
- **Stability, linked lists, or disk.** Merge sort: it is stable, merges sorted pieces, and is the basis of external sorting.
- **Large records.** Sort indices or links, not the records themselves; linked merge sort is an example.
- **Fixed-length integer keys.** Radix sort: *d* passes, each $O(n + r)$.
- **Keys in a small range.** Counting sort.

20.11 Common mistakes

1. Forgetting the `j == L` test in the `j` loop of `partition`: if the pivot is the smallest, the scan runs off the start of the array.
2. Always choosing `a[R]` as the pivot on sorted input: the depth becomes n and the time n^2. The remedy is median-3 or a random pivot.
3. Pushing the parts without looking at their sizes in the non-recursive version: the stack grows to $O(n)$ and the fixed-size array overflows.
4. Giving the `SWAP` arguments in reverse in median-3: the larger item goes to `a[L]` and partitioning breaks.
5. Calling `malloc` for `b[]` on every call (slow) or never allocating it (undefined behavior); forgetting the `NULL` check and `free`.

6. Writing `i1` instead of `i2` in the `else` branch of the linked `merge`; relying on the order of evaluation of arguments.
7. Using one-based formulas with a zero-based array in `heapify`.
8. Writing `return x - y;` in a `qsort` comparator, or using an inconsistent comparator.
9. Adding a node to the front of a bucket in radix sort: distribution becomes unstable and the result is wrong.

20.12 Exercises and solutions

The answers are the same as those in the slides; here the solution steps are also given and the code was verified by running it.

1. For *[5, 9, 1, 7, 3]*, what are the array and the return value after `partition(a, 0, 4)`?

Solution. The pivot is $v = a[4] = 3$. `i` stops at `a[0] = 5` ($5 \geq 3$). `j` passes 7 and stops at `a[2] = 1` ($1 \leq 3$). $i = 0 < j = 2$: swap, the array is *[1, 9, 5, 7, 3]*. Then `i` stops at `a[1] = 9`; `j` passes 9 and stops at `a[0] = 1`. $i = 1 \geq j = 0$: crossed. The pivot is swapped with `a[1]`. The array is *[1, 3, 5, 7, 9]* and the return value is 1.

2. How many `partition` calls does quicksort (`quicksort1`) make on a sorted array of 8 items? What is the recursion depth?

Solution. The pivot is the largest item of the range every time and stays in place. Ranges of sizes 8, 7, …, 2 are partitioned: 7 `partition` calls. Because each partition is done inside the previous one, the number of nested, waiting frames that call `partition` is $n - 1 = 7$. The number of comparisons is $9 + 8 + \ldots + 3 = 42$.

3. How many `merge` calls does merge sort make for 16 items? What is the largest number of comparisons?

Solution. The frame tree has 16 leaves and $16 - 1 = 15$ internal nodes: 15 `merge` calls. The `merge` in this note makes exactly one comparison per placement. At each of the $\lg 16 = 4$ levels of the tree, 16 items in total are merged: $4 \times 16 = 64$ comparisons. The count is independent of the input; the instrumented run also gave 64.

4. For *[4, 10, 3, 5, 1]* (`a[1..5]`), what is the array after heap construction?

Solution. Start at $[5/2] = 2$. `heapify(a, 2, 5)`: $t = 10$, children 5 and 1; $10 \geq 5$, no change. `heapify(a, 1, 5)`: $t = 4$, children 10 and 3; 10 moves up. The hole descends to 2; children 5 and 1, $4 < 5$: 5 moves up. The hole descends to 4; it has no children and 4 is written there. The result is *[10, 5, 3, 4, 1]*.

5. What order does radix sort give after the units pass on the list *170, 45, 75, 90, 802, 24, 2, 66*?

Solution. The units digits are 0, 5, 5, 0, 2, 4, 2, 6 in turn. The buckets: 0: 170, 090; 2: 802, 002; 4: 024; 5: 045, 075; 6: 066. Stable distribution and collection give *170 090 802 002 024 045 075 066*. The next two passes produce the lists *802 002 024 045 066 170 075 090* and *002 024 045 066 075 090 170 802*.

6. Why is heap sort not stable? Give a small example.

Solution. The array *[2a, 2b, 1]* is already a max-heap. In the first round the 2a at the root is moved to the end: *[1, 2b, 2a]*. `heapify` brings 2b to the root, and in the second round 2b goes to the second position. The result is *[1, 2b, 2a]* and the order of the equal keys has changed. Heap sort swaps items between distant positions; these distant swaps do not preserve the relative order of equal items.

20.13 Summary

- **Quicksort.** `partition` puts the pivot in its final place and splits the range in two. Best case $n \lg n$, average $\approx 1.39\, n \lg n$, worst case n^2 comparisons; in place, unstable. The pivot is the sentinel of `i`; `j` needs `j == L`.
- **Average case.** The recurrence is brought to the form $C_n/(n+1) = C_{n-1}/n + 2/(n+1)$ by the subtraction trick and solved with harmonic numbers.
- **Improvements.** If the larger part is pushed first, at most $\lfloor \lg n \rfloor + 1$ pairs wait on the explicit stack. Median-3 removes the n^2 on sorted input; small parts are left to insertion sort. Introsort brings the worst case down to $O(n \lg n)$ with heap sort.
- `qsort`. It sorts any type with `void *` and a function pointer; the comparator must be consistent and must not overflow. There is no stability guarantee.
- **Merge sort.** $\Theta(n \lg n)$ in every case; $2n - 1$ calls, $n - 1$ merges, $O(n)$ extra memory. The linked version moves no records and changes only `link` indices.
- **Heap sort.** Heap construction is $O(n)$ with at most $2^k - k - 1 \le 2n$ shifts, sorting is $O(n \lg n)$; extra memory $O(1)$, unstable.
- **Radix sort.** LSD to MSD with stable buckets, $O(d(n + r))$; for non-negative integers and a small range, without comparisons.

20.14 Key terms

These are the standard English terms for the concepts of this chapter.

divide and conquer quicksort partition pivot invariant sentinel recurrence relation lower bound stack frame stack overflow median-of-three insertion sort hybrid sort introsort void pointer function pointer record stable (sort) in-place sort merge sort heap sort max-heap bottom-up heap construction complete binary tree radix sort bucket counting sort key

20.15 Try It Yourself

1. `partition(a, 0, 7)` is called on the array *[2, 8, 7, 1, 3, 5, 6, 4]*. What happens to the array, and what value is returned?

   Answer

   The pivot is 4. `i` stops at 8 and `j` at 3: swap, *[2, 3, 7, 1, 8, 5, 6, 4]*. Then `i` stops at 7 and `j` at 1: swap, *[2, 3, 1, 7, 8, 5, 6, 4]*. Then `i` stops at 7 (position 3) and `j` at 1 (position 2), and they cross. The pivot goes to position 3: *[2, 3, 1, 4, 8, 5, 6, 7]*, return 3.

2. How many `partition` calls does `quicksort1` make on the reverse-sorted array *[6, 5, 4, 3, 2, 1]*? Why is this input also bad?

   Answer

   5 calls. Each pivot falls at one end of the range and each split separates only one item; the depth is $n - 1$ and the work is of order n^2.

3. Compute C_2 and C_3 from the average-case recurrence.

   Answer

   $C_2 = 3 + (2/2)(C_0 + C_1) = 3$. $C_3 = 4 + (2/3)(C_0 + C_1 + C_2) = 4 + 2 = 6$. The closed form also gives, for $n = 3$, $2 \cdot 4 \cdot H_4 - 32/3 = 50/3 - 32/3 = 6$.

4. While `quicksort2` sorts an array of 1000 items, at most how many *(L, R)* pairs can be on the stack at one time? Why is `YMAX = 64` enough?

   Answer

   $\lfloor \lg 1000 \rfloor + 1 = 10$ pairs, that is 20 `int` s. Even for $n < 2^{31}$ the bound is $2 \cdot 31 = 62$ `int` s, which does not exceed 64. The condition for this is that the larger part is pushed first.

5. With *a = [2, 9, 4, 5]*, `merge(a, b, 0, 1, 3)` is called. What is `b` after copying? What does `a` become?

   Answer

   The left half is copied as is and the right half in reverse: *b = [2, 9, 5, 4]*. Merging yields the order 2, 4, 5, 9: *a = [2, 4, 5, 9]*.

6. For 8 items, how many `mergesort` calls, how many `merge` calls, and how many comparisons does

merge sort make?

> **Answer**
>
> *2n − 1 = 15* calls, *n − 1 = 7* merges, and *8 · lg 8 = 24* comparisons.

7. Bottom-up heap construction is applied to `a[1..8] = [3, 1, 4, 1, 5, 9, 2, 6]`. What is the result?

> **Answer**
>
> *[9, 6, 4, 1, 5, 3, 2, 1]*. `heapify` is called for 4, 3, 2, and 1, with 5 shifts in total.

8. Radix sort is applied to the list *512, 037, 208, 099*. What is the list after the tens pass?

> **Answer**
>
> The units pass gives *512 037 208 099* (2, 7, 8, 9). The tens pass, on the digits 1, 3, 0, 9, gives *208 512 037 099*. After the hundreds pass the list is *037 099 208 512*.

9. How do you write a comparison function that does not overflow and sorts an `int` array from largest to smallest with `qsort`?

> **Answer**
>
> `int descending(const void *p, const void *q) { int x = *(const int *)p, y = *(const int *)q; return (y > x) - (y < x); }`. In `ascending`, `x` and `y` are swapped. If `return y - x;` were written, large differences would overflow.

20.16 Further reading

- R. Sedgewick, *Algorithms in C*, Parts 1–4, Chapters 7–10. It is the source of this note's `partition`, `quicksort2`, median-3, and bitonic `merge` code; it compares the versions of quicksort with measurements.
- E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, Chapter 7. It presents linked merge sort, heap sort with `adjust`, and bucket radix sort with the examples used in this note.
- T. H. Cormen et al., *Introduction to Algorithms*, Chapters 6–8. It gives detailed proofs for heap sort, the probabilistic analysis of quicksort, and linear-time sorting.
- D. Musser, “Introspective Sorting and Selection Algorithms,” 1997. The paper in which introsort was introduced; it also examines inputs that force median-3 to n^2.

20.17 References

1. R. Sedgewick, *Algorithms in C*, Parts 1–4, Chapters 7–10 (quicksort, mergesort, priority queues and heapsort, radix sorting).
2. E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, Chapter 7.
3. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, Chapters 6–8.
4. D. Musser, “Introspective Sorting and Selection Algorithms”, 1997.
5. ISO/IEC 9899 (the C standard), definition of `qsort` (§7.22.5.2).

APPENDIX A

Glossary of Terms

The terms used in the book, each with a short definition. The number next to a term is the chapter where it is explained in detail. A searchable version of the glossary is on the course page.

A

abstract data type (ADT) 1
A type defined by its values and operations alone, independently of how they are implemented.

Ackermann function 16
A very fast-growing recursive function whose inverse, α(n), bounds the cost of union-find.

activity network 18
A directed graph that shows the tasks of a project and the precedence relations among them.

activity on edge (AOE) network 18
An activity network in which the tasks are edges and the events are vertices.

activity on vertex (AOV) network 18
An activity network in which the tasks are vertices and the precedence relations are edges.

address-of operator (&) 4
The unary operator that yields the address of an object.

adjacency list 15
A representation that keeps, for each vertex, a linked list of its neighbors.

adjacency matrix 15
An n×n matrix in which a[i][j] = 1 if there is an edge between i and j.

adjacency multilist 15
A representation of an undirected graph in which each edge is stored once but appears in the edge lists of both of its endpoints.

adjacent 15
Two vertices that are joined by an edge.

aggregate analysis 8
An analysis that counts the total cost of all the steps together rather than of a single step; one way of doing amortized analysis.

algorithm 1
A method that solves a problem in a finite number of precise and effective steps.

alignment 2
The compiler's placement of fields at addresses that are multiples of a given size; it affects the size of structures.

all-pairs shortest paths 18
The problem of finding a shortest path between every pair of vertices.

amortized analysis 1
Finding the cost per operation by dividing the total cost of a sequence of operations by the number of operations.

ancestor 10
Any node on the path from the root to a given node.

array 2
A structure whose elements have the same type, are stored consecutively in memory, and are accessed by index.

arrow operator (->) 1
The operator that accesses a structure field through a pointer: p->field is the same as (*p).field.

associativity 8
The rule that decides whether operators of equal precedence are applied from left to right or from right to left.

asymptotic notation 1
A notation that compares the growth rates of functions for large n by discarding constant factors.

available space list 4
A list that keeps freed nodes so that they can be reused (avail).

average case 1
The expected running time over a given distribution of inputs.

AVL tree 11
A search tree that keeps the height difference of the two subtrees of every node at most 1.

B

back edge 15
In DFS, an edge to an already marked ancestor other than the parent; it shows that the graph contains a cycle.

backtracking 3
A search method that extends a partial solution step by step and undoes the last choice when it reaches a dead end.

balance factor 11
The height of the left subtree minus the height of the right subtree.

balanced parentheses 6
Checking with a stack whether every opening parenthesis is closed by one of the right type and in the right order.

balanced tree 10
A search tree whose height is kept at O(lg n).

band matrix 2
A matrix whose nonzero elements lie in a band around the main diagonal.

base case 1
The case in which a recursion returns a result directly, without calling itself.

Bellman-Ford algorithm 18
A shortest-path algorithm that relaxes all edges n − 1 times and allows negative weights.

best case 1
The running time for the input that needs the fewest steps among inputs of the same size.

Big O 1
An upper bound: f(n) = O(g(n)) if f(n) ≤ c·g(n) for all n ≥ n_0.

Big Omega 1
A lower bound: f(n) = Ω(g(n)) if f(n) ≥ c·g(n) for all n ≥ n_0.

Big Theta 1
A tight bound: f(n) = Θ(g(n)) if f(n) is both O(g(n)) and Ω(g(n)).

binary search 3
Searching a sorted array by halving the range at every step; O(lg n).

binary search tree (BST) 11
A binary tree in which, at every node, the keys in the left subtree are smaller and those in the right subtree are larger.

binary tree 9
A tree in which every node has at most two children, a left and a right one.

bipartite graph 15
A graph whose vertices can be split into two groups so that every edge joins the two groups.

bottom-up 1
An approach that first builds small pieces and then combines them into the whole.

bottom-up heap construction (heapify) 13
Turning an array into a heap in O(n) time by sifting down from the last internal node to the root.

breadth-first search (BFS) 7
A traversal that visits vertices in order of their distance from the start, using a queue.

brute force 1
An approach that tries every possible solution one by one.

B-tree 11
A balanced m-way search tree that stores many keys per node and keeps all leaves at the same level.

bubble sort 19
A sort that compares neighboring elements and swaps those that are out of order, moving the large ones to the end.

bucket 14
The compartment of a hash table that corresponds to one index.

bucket sort 19
A method that distributes elements into buckets by value range and then sorts each bucket separately.

buffer 7
Memory that temporarily holds data between a producer and a consumer.

C

cache 6
A small, fast memory in which the processor keeps frequently used data; the reason arrays are faster than linked structures.

call by reference 11
Passing the address of a variable so that the function can modify it.

call by value 6
Passing a copy of the argument; in C all arguments are passed this way.

call stack 1
The stack that holds the frames of the active function calls.

calloc 2
A function that allocates a block of n elements and sets it to zero.

capacity 1
The number of elements a dynamic array can hold without reallocating.

child 3
A node directly below a given node.

circular linked list 4
A linked list in which the last node points to the first node.

circular queue 7
A queue implementation in which an index that reaches the end of the array wraps around to the beginning.

collapsing rule 16
Linking the nodes on the path directly to the root while finding a set.

collision 14
Two different keys falling into the same bucket.

column-major order 2
Storing a multidimensional array column by column.

compaction 5
Shifting the polynomials still in use to the beginning of the array when it is full, so as to reclaim the freed space.

complete binary tree 10
A binary tree that is full except possibly at the last level, which is filled from left to right.

complete graph 15
A graph in which every pair of vertices is joined by an edge; with n vertices it has n(n − 1)/2 edges.

compound literal 8
A C99 expression that creates a temporary, unnamed structure object and fills in its fields, for example (element){row, col, dir}.

compressed sparse column (CSC) 5
The column-major counterpart of CSR: a sparse matrix is compressed by columns instead of rows.

compressed sparse row (CSR) 5
A representation that stores a sparse matrix with arrays of values, column indices, and row start positions.

connected 15
Having a path between every pair of vertices.

connected component 15
One of the largest connected subgraphs of a graph.

const qualifier 1
A keyword declaring that a variable, or the object a pointer points to, will not be modified.

counting sort 5
A sort that counts how many times each key value occurs and then places the elements directly into position.

critical path 18
The longest path from start to finish; it determines the shortest possible duration of the project.

cut 17
A partition of the vertices of a graph into two nonempty groups, S and V − S; an edge with one end in S and the other in V − S crosses the cut.

cycle 4
A path that returns to the vertex it started from; it occurs in linked lists as well as in graphs.

cycle detection 4
Finding out whether a linked structure contains a path that leads back to itself.

D

dangling pointer 2
A pointer that still points to an object that has been freed or whose lifetime has ended.

decision tree 19
A binary tree that shows all possible outcomes of the comparisons made by a comparison-based algorithm.

decrease key 13
Decreasing the key of an element in a heap and restoring its position; used in Dijkstra's and Prim's algorithms.

degree 10
In a tree, the number of children of a node; in a graph, the number of edges incident to a vertex.

depth 10
The largest level in a tree; the textbook also uses this name for the height of a tree.

depth-first search (DFS) 6
A traversal that goes to a neighbor and descends as deep as possible, using a stack or recursion.

depth-first spanning tree 15
A spanning tree made of the edges along which vertices are first entered during a DFS.

deque (double-ended queue) 7
A queue in which insertion and removal are possible at both ends.

dequeue 7
Removing the element at the front of a queue.

dereference (*) 4
The unary operator that reaches the object a pointer points to.

descendant 10
Any node below a given node.

dictionary 14
An ADT that stores key–value pairs and supports lookup by key.

Dijkstra's algorithm 18
A greedy algorithm that finds shortest paths for nonnegative weights by finalizing the nearest vertex at each step.

direct addressing 14
Using the key itself as an array index.

directed acyclic graph (DAG) 18
A directed graph that contains no directed cycle.

directed graph (digraph) 15
A graph whose edges each have a tail and a head.

disjoint sets 16
Sets with no common element; every element belongs to exactly one set.

divide and conquer 20
An approach that splits a problem into smaller subproblems and combines their solutions.

division method 14
The hash function h(k) = k mod m, where m is usually chosen to be prime.

double hashing 14
Determining the probe step with a second hash function.

double rotation 11
Two single rotations performed one after the other in opposite directions.

doubling 2
Doubling the capacity when the array is full; this makes the amortized cost of appending O(1).

doubly linked list 4
A linked list in which every node points to both its predecessor and its successor.

down-heap 13
Moving the element at the root down by swapping it with its larger (or smaller) child.

dummy activity 18
A task of duration zero that is added only to express a precedence relation.

dynamic array 1
An array that is enlarged with realloc when it becomes full.

dynamic memory allocation 2
Requesting memory at run time with malloc, calloc, or realloc.

E

edge 15
A connection between two vertices.

enqueue 7
Adding an element at the rear of a queue.

enumeration (enum) 2
A C type consisting of named integer constants.

equivalence class 16
Each of the disjoint sets into which a reflexive, symmetric, and transitive relation partitions the elements.

event 18
In an activity-on-edge network, the moment that a vertex represents: the time at which all activities entering it have finished.

exponential 1
Growth such as 2^n, which doubles whenever one more element is added to the input.

expression evaluation 6
Computing the value of an arithmetic expression while respecting operator precedence.

expression tree 10
A binary tree whose internal nodes are operators and whose leaves are operands.

external sorting 19
Sorting in which the data does not fit in memory and auxiliary storage such as a disk is used.

F

fast transpose 5
A method that transposes a sparse matrix in O(columns + terms) time by first counting the elements in each column.

Fibonacci heap 13
An advanced heap that performs decrease-key in amortized O(1) time.

FIFO (first in, first out) 7
The order in which the element inserted first is removed first; the rule of a queue.

find 16
Determining which set an element belongs to by returning the root of that set.

Floyd-Warshall algorithm 18
An algorithm that finds all-pairs shortest paths in $O(n^3)$ time by adding intermediate vertices one at a time.

folding 14
Producing a hash value by splitting the key into parts and adding the parts.

forest 17
A collection of disjoint trees.

free 1
The function that returns a block allocated with malloc to the system.

free float 18
The slack that an activity can use without delaying the earliest start time of its successors.

front 7
The end of a queue that holds the next element to be removed.

function pointer 20
A pointer that holds the address of a function; this is how a comparison function is passed to qsort.

G

garbage collection 9
Automatically finding and reclaiming nodes that can no longer be reached.

generalized list 9
A list whose elements may be atoms or other lists.

global variable 6
A variable defined at file level that can be accessed from all functions.

graph 10
A structure consisting of vertices and the edges that join them: G = (V, E).

greedy method 17
An approach that makes the choice that looks best at the moment at every step and never goes back on it.

H

hash flooding 14
An attack in which the attacker sends many keys that fall into the same bucket, degrading hash table lookups to O(n) and costing the server $O(n^2)$ work.

hash function 14
A function that converts a key into an index smaller than the table size.

hash table 14
A table that maps a key directly to a bucket by a hash function; average access time is O(1).

head 4
The pointer that points to the first node of a linked list.

header file 1
A .h file that carries type and function declarations and is added with #include.

header node 4
A node that holds no data and sits at the start of the list, removing the special case of an empty list.

heap 13
A complete binary tree in which the key of every node is larger (or smaller) than the keys of its children.

heap memory 2
The region of memory allocated with malloc that lives until it is freed.

heap sort 13
An in-place algorithm that sorts an array by turning it into a heap and repeatedly moving the root to the end.

height 10
The number of levels from a node down to its farthest leaf; the balance of an AVL tree is measured by it.

heuristic 3
A method that approaches a solution quickly with a clever choice rule without guaranteeing the best one, for example Warnsdorff's rule in the knight's tour.

home address 14
The index that a key's hash function gives, that is, h(k).

Horner's method 2
Evaluating a polynomial by nested multiplication: $a_0 + x(a_1 + x(a_2 + \ldots))$.

hybrid sort 20
A method that combines several sorting algorithms, for example quick sort and insertion sort.

I

implementation 1
The concrete writing of an ADT with a particular data structure and code.

in place 2
Working with only a constant amount of extra memory besides the input.

incident 15
The relationship between an edge and its endpoints: edge (v0, v1) is incident on vertices v0 and v1.

include guard 1
The #ifndef … #define … #endif pattern that prevents a header from being included twice in the same file.

incoming precedence (icp) 8
The precedence an operator has at the moment it is read from the input.

incomplete type 1
A type whose size is not yet known; only a pointer to it can be defined.

in-degree 15
In a directed graph, the number of edges coming into a vertex.

index 2
An integer that specifies the position of an element in an array.

infix 8
The notation in which the operator is written between the two operands: a + b.

inorder 3
A traversal that visits the left subtree, then the root, then the right subtree.

inorder predecessor 12
The node visited immediately before a given node in an inorder traversal.

inorder successor 12
The node visited immediately after a given node in an inorder traversal.

in-place sort 13
A sort that uses only a constant amount of extra memory.

insertion sort 19
A sort that inserts each element by shifting it to its correct place in the sorted part before it.

in-stack precedence (isp) 8
The precedence an operator has while it waits on the stack.

interface 1
The types and function declarations a module exposes to the outside; in C it lives in the header file.

internal node 10
A node that has at least one child.

internal sorting 19
Sorting in which all the data resides in main memory and every element is accessed in constant time.

introsort 20
A quick sort that switches to heap sort when the recursion depth limit is exceeded.

invariant 20
A property of a loop or structure that remains true at every step.

inverse Ackermann function α(n) 16
An extremely slowly growing function that in practice never exceeds 4.

inversion 19
A pair of elements in an array with i < j but a[i] > a[j].

isomorphic 10
Able to be made the same structure by a one-to-one matching of their nodes.

K

key 5
The value used to search for, sort, or distinguish elements.

knight's tour 3
The problem of visiting every square of the board exactly once with a chess knight.

Kruskal's algorithm 16
An MST algorithm that tries edges in increasing order of cost and takes those that do not form a cycle.

L

leaf 3
A node that has no children.

least significant digit (LSD) 20
In multikey sorting, the key or digit with the lowest priority; LSD radix sort starts from this digit.

level 10
The distance of a node from the root; the root is at level 1.

level-order 7
Visiting nodes level by level from left to right, using a queue.

LIFO (last in, first out) 6
The order in which the element inserted last is removed first; the rule of a stack.

linear probing 14
On a collision, looking at the next bucket in sequence.

linear (sequential) search 1
Searching by comparing elements one by one from beginning to end; O(n).

link 4
A pointer field that connects a node to another node.

linked list 1
A sequence of elements scattered in memory, in which each node holds the address of the next one.

list ADT 9
A type for a sequence of elements that supports insertion, deletion, and access by position.

load factor 14
The ratio of the number of elements in the table to the number of buckets: α = n / m.

look-up table 8
An array in which values that would otherwise be computed with a chain of conditions are stored in advance, for example the row and column increments of the eight directions in a maze.

lower bound 19
The least time that any algorithm solving a problem must spend; Ω(n lg n) for comparison-based sorting.

lower triangular matrix 2
A square matrix in which all elements above the main diagonal are zero.

M

macro 1
A name defined with #define that is replaced as text before compilation.

magic square 1
An n×n table of numbers in which every row, column, and main diagonal has the same sum.

malloc 1
A function that allocates a block of memory of the requested number of bytes and returns NULL on failure.

mapping 2
A relation that assigns a value to each index; in the abstract view, this is what an array is.

mark and sweep 9
A two-phase garbage collection: elements reachable from the root pointers are marked, then memory is scanned and the unmarked elements are reclaimed; it also collects cycles.

max-heap 13
A heap in which every node is greater than or equal to its children; the largest element is at the root.

maze 6
A grid of walls and paths; a route from entrance to exit is found by backtracking with a stack.

median-of-three 20
Choosing the pivot as the median of the first, middle, and last elements.

memory leak 4
Memory that can no longer be reached but has not been freed.

merge 5
Combining two sorted arrays or lists into a single sorted array.

merge sort 20
An algorithm that splits an array in two, sorts the parts, and merges the sorted parts; O(n lg n).

mid-square 14
A hash function that squares the key and uses the middle r bits of the result as the index; the table size m is the r-th power of 2.

min-heap 13
A heap in which every node is less than or equal to its children; the smallest element is at the root.

minimum spanning tree (MST) 15
A spanning tree whose edge costs have the smallest sum.

modular arithmetic 7
Computation with remainders; circular queues and hash functions rely on it.

monotonic stack 8
A stack whose element values are kept in decreasing order from bottom to top; it solves problems such as "next greater element to the right" in O(n).

Morris traversal 12
An inorder traversal with no stack and O(1) extra memory that temporarily sets up and removes threads.

most significant digit (MSD) 20
In multikey sorting, the key or digit with the highest priority (K0); for example, the leftmost digit of a number.

multidimensional array 2
An array whose elements are accessed with more than one index.

multigraph 15
A graph that may have more than one edge between two vertices; the Königsberg graph is an example.

multiplication method 14
Finding the index by multiplying the fractional part of k·A by m.

multivariate polynomial 9
A polynomial with more than one variable; it is represented by a generalized list.

N

negative cycle 18
A cycle whose edge weights sum to a negative value; it makes the shortest path undefined.

node 3
A structure that holds data and one or more link fields.

non-comparison sort 20
A sort that does not compare elements with one another and is therefore not bound by the n lg n lower bound, such as radix sort.

NP-complete 3
The class of problems such that if a polynomial-time algorithm is found for any one of them, all can be solved in polynomial time; SAT is the first member of this class.

n-queens problem 3
The problem of placing n queens on an n×n board so that no two threaten each other.

null character ('\0') 2
The character with value zero that marks the end of a string.

null pointer (NULL) 1
A pointer that points to no object; it represents an empty list and an empty subtree.

O

opaque type 1
A structure type of which only the name is declared and whose fields are hidden from the user; the C counterpart of an ADT.

open addressing 14
On a collision, looking for another empty position inside the table.

operand 8
In an expression, a value to which an operation is applied: variables such as a and b, or constants such as 4 and 2.

operator 8
In an expression, a symbol such as +, −, *, / that specifies the operation to perform.

operator precedence 8
The rule that determines which operator is applied first in an expression.

optimal substructure 18
The property that every intermediate part of a shortest path is itself a shortest path between its own endpoints.

orthogonal list 15
A representation in which every edge node appears in both a row list and a column list.

out-degree 15
In a directed graph, the number of edges leaving a vertex.

overflow 6
An attempt to insert into a full structure.

P

padding 2
Meaningless bytes the compiler places between structure fields for alignment; this is why structures are not compared byte by byte.

parent 10
The node directly above a given node.

parent array 16
An array that stores the parent of each element in the tree; the root holds a negative value.

partition 20
Splitting an array into two parts, the elements smaller than the pivot and those larger.

path 15
A sequence of consecutively adjacent vertices.

path compression 16
The general name of the collapsing rule; it shortens later find operations.

perfect binary tree 10
A binary tree with all levels full, having $2^k - 1$ nodes at depth k; the textbook (Horowitz) calls this a full binary tree.

performance analysis 1
Estimating the time and memory requirements of a program independently of the machine.

performance measurement 1
Running a program on a particular machine and compiler and measuring its actual running time; it depends on the machine.

permutation 1
One of the possible arrangements of the elements of a set; n elements have n! permutations.

pivot 20
The element chosen in quick sort to divide the array in two.

pointer 2
A variable that holds the memory address of an object.

pointer arithmetic 2
Adding an integer to or subtracting it from a pointer; the unit is one element of the pointed-to type, not a byte.

pointer to pointer 9
A pointer that holds the address of a pointer; it is used to change the caller's pointer.

polynomial 5
A sum of terms, each made of a coefficient and an exponent.

pop 6
Removing the element at the top of a stack.

postfix (reverse Polish notation) 8
The notation, which needs no parentheses, in which the operator is written after the operands: a b +.

postorder 3
A traversal that visits the left subtree, then the right subtree, then the root.

predecessor 12
The node that comes immediately before a given node in the inorder sequence.

prefix (Polish notation) 8
The notation in which the operator is written before the operands: + a b.

preorder 9
A traversal that visits the root, then the left subtree, then the right subtree.

preprocessor 1
The stage that processes directives such as #include and #define before compilation.

primary clustering 14
In linear probing, the formation of long consecutive blocks of occupied buckets.

Prim's algorithm 17
An MST algorithm that starts from one vertex and adds the vertex joined to the tree by the cheapest edge.

priority queue 5
A queue from which the element with the highest (or lowest) priority is always removed.

probe 14
In open addressing, checking whether a bucket is empty.

probe sequence 14
In open addressing, the order of slots examined for a key, h(k, i).

program step 1
A meaningful piece of code whose running time is independent of the input size; the unit of step counting.

push 4
Adding an element to the top of a stack.

Q

quadratic probing 14
On a collision, looking at buckets i^2 steps away.

queue 7
A linear structure in which insertion is done at the rear and removal at the front.

quick sort 19
An algorithm that partitions an array around a pivot element and sorts the two parts recursively.

R

radix sort 20
Sorting numbers digit by digit, starting from the least significant digit, with a stable sort.

random access 4
Being able to access any element of an array in constant time, that is O(1), regardless of the array's length.

realloc 1
A function that enlarges or shrinks an allocated block, moving it if necessary.

rear 7
The end of a queue where a new element is inserted.

record 19
A unit of data made of one or more fields that is moved as a whole during sorting.

recurrence relation 3
An equation that defines a function in terms of its own smaller values, such as T(n) = 2T(n/2) + n.

recursion 1
A function calling itself with a smaller input.

recursive 1
Calling itself; using recursion.

red-black tree 11
A search tree that keeps its height O(lg n) through rules on node colors.

reference count 9
The number of pointers that point to a node; when it drops to zero, the node is returned.

rehashing (resizing) 14
Enlarging the table when the load factor grows and hashing all the keys again.

relaxation 18
Updating a distance value when a shorter path is found through an edge.

return address 3
The address of the instruction at which execution continues when a call ends.

reverse-delete 17
An MST algorithm that starts with all edges and tries them in decreasing order of cost, deleting each one that does not disconnect the graph; the reverse of Kruskal's.

reversing a list 4
Reversing the order of a list by changing the direction of its links; in place and O(n).

root 3
The topmost node of a tree, which has no parent.

rotation 11
An operation that changes the local structure of a tree while preserving the search order.

row-major order 2
Storing a multidimensional array row by row consecutively; the layout of C.

S

satisfiability (SAT) 3
The problem of whether some assignment of variables makes a logical formula true.

secondary clustering 14
In quadratic probing, keys with the same home address follow the same probe sequence and block one another.

selection sort 1
A sort that at each step finds the smallest of the remaining elements and puts it in the next position.

self-balancing tree 11
A tree that restores its balance by itself through rotations after insertions and deletions.

self-loop 15
An edge that joins a vertex to itself; simple graphs do not contain them.

self-referential struct 2
A structure with a pointer field of its own type; the basis of linked nodes.

sentinel 1
A special value or node placed at the end so that a separate boundary check in a loop becomes unnecessary.

separate chaining 14
Keeping the keys that fall into each bucket in a linked list; in the slides it is simply called “chaining”.

Shell sort 19
A sort that applies insertion sort to elements far apart, using decreasing gaps.

short-circuit evaluation 2
In the && and || operators, not evaluating the remaining operand once the result is known.

shortest path 8
A path between two vertices whose edge costs have the smallest sum.

sibling 3
Children of the same parent.

simple uniform hashing 14
The assumption that every key falls into each of the m buckets with probability 1/m, independently of the other keys.

singly linked list 4
A linked list in which each node points only to the next node.

sizeof 1
The operator that gives the size of a type or object in bytes at compile time.

skewed tree 10
A tree all of whose nodes extend in a single direction, behaving like a list.

slack 18
The difference between the latest and earliest time of a task; zero for critical tasks.

slot 14
Each cell of a hash table; used synonymously with bucket.

Sollin's (Borůvka's) algorithm 17
An MST algorithm that in each round selects the cheapest outgoing edge of every component.

sorting 1
Putting elements into ascending or descending order by their keys.

space complexity 1
The amount of memory a program needs as a function of the input size.

spanning forest 17
In a graph that is not connected, the collection of spanning trees of its components.

spanning tree 15
A subgraph of a connected graph that joins all its vertices without cycles; it has n − 1 edges.

sparse matrix 5
A matrix most of whose elements are zero; only the nonzero ones are stored.

splay tree 11
A search tree that moves the accessed node to the root by rotations.

stable (sort) 19
A sort that preserves the relative order of elements with equal keys.

stack 6
A linear structure in which insertion and removal take place only at the top end.

stack frame (activation record) 1
A record that holds the parameters, local variables, and return address of a call; the text also calls it simply a “frame”.

stack overflow 3
The call stack exceeding its allotted memory; it usually happens with very deep or infinite recursion.

string 2
In C, a char array terminated by '\0'.

struct 1
A C type that gathers fields of different types into a single record.

subgraph 15
A graph whose vertices and edges are subsets of the vertices and edges of a graph.

subset 3
A set formed from some of the elements of a set; n elements have 2^n subsets.

subtree 9
A tree made of a node and all its descendants.

successor 12
The node that comes immediately after a given node in the inorder sequence; used in deletion and in threaded trees.

swap 1
Exchanging the values of two variables.

synonyms 14
Keys that fall to the same home address, that is, keys that collide.

system life cycle 1
The development process consisting of the requirements, analysis, design, coding, and verification phases.

T

tagged union 2
A structure that records with an enum tag which field is valid.

tail (of a generalized list) 9
In a generalized list, the list that remains after the first element is removed.

tail recursion 3
Recursion in which the recursive call is the last operation of the function; it is easily converted into a loop.

thread 10
In a threaded tree, a pointer that points to the predecessor or successor in place of an empty child.

threaded binary tree 10
A binary tree whose null pointers point to the inorder predecessor and successor.

time complexity 1
The number of steps a program executes as a function of the input size.

token 8
An item processed as a single unit while parsing an expression: an operand, an operator, or a parenthesis.

tombstone 14
In open addressing, a mark placed in a deleted bucket so that a search does not stop there.

top 6
The end of a stack where the most recently inserted element is.

top-down 1
A design approach that first treats the problem as a whole and then splits it into ever smaller parts.

topological order 15
An ordering of vertices in which the tail of every edge comes before its head.

tortoise and hare 4
Floyd's method of finding a cycle with two pointers, one advancing one step and the other two steps at a time.

total float 18
The difference between the latest and earliest start times of an activity; it is shared among the activities on the same path and can be spent only once along the path.

Towers of Hanoi 3
The problem of moving disks among three pegs according to the rules; it takes $2^n - 1$ moves.

transpose 2
The matrix obtained by interchanging the rows and columns of a matrix.

traversal 2
Visiting every element of a structure once in a particular order.

tree 1
A hierarchical structure with no cycles that starts from a root and branches into subtrees.

triple 5
In a sparse matrix, the (row, column, value) record that holds one nonzero element.

typedef 1
A C declaration that gives a new name to an existing type.

U

undefined behavior 2
A situation for which the C standard does not determine the result, for example access beyond an array's bounds; the program may crash or appear to work correctly.

underflow 6
An attempt to remove an element from an empty structure.

undirected graph 15
A graph whose edges have no direction.

undo 6
Reversing the most recent operations in reverse order; implemented with a stack.

union 16
The operation of merging two sets into one.

union by rank 16
In a union, using the height bound (rank) instead of the tree weight.

union (C) 2
A C type whose fields all share the same memory region; only one field is valid at a time.

union-find 16
A disjoint-set structure that supports the union and find operations.

universal hashing 14
Choosing the hash function at random at the start from a family in which two different keys collide with probability at most 1/m.

up-heap 13
Moving an inserted element up to its proper place by swapping with its parent.

V

vertex 15
An element of a graph; the same term as node in a tree is used.

void pointer (void *) 20
A pointer of unspecified type; in generic routines such as qsort it is cast to any type.

W

weighted graph (network) 15
A graph whose edges are assigned a cost or length.

weighting rule 16
In a union, attaching the tree with fewer nodes under the tree with more nodes; also called union by size.

worst case 1
The running time for the input that needs the most steps among inputs of the same size.

wrap-around 7
In a circular queue, the index returning to the start after the last cell, by (rear + 1) % MAX.

APPENDIX B

Prerequisites for Data Structures and Algorithms with C

In this course every structure is built in C with memory managed by hand: a node is defined with `struct`, allocated with `malloc`, linked with pointers, and given back with `free` when it is no longer needed. You should therefore be able to read C's types, pointers, arrays, and dynamic memory comfortably before starting. Comparing costs also takes a little mathematics: logarithms, summation formulas, and induction.

Each card gives a short summary of the topic, a small C example, and a check question; the answer is in the collapsible box under the question. The links go to the slides where the topic is used in the course.

Suggested order of preparation

Types and functions · Pointers · Arrays and strings · struct · Dynamic memory · Recursion · Logarithms and sums · Big O

C fundamentals

TY

Basic Types and sizeof

int, char, double · integer division · number of bytes

In C every variable has a type and occupies a definite number of bytes in memory; `sizeof` gives that number at compile time. Division between two integers is integer division: the fractional part is discarded. Array indices, the parent index in a heap (`i / 2`), and the midpoint in binary search all rely on this rule.

```
int a = 7, b = 2;
printf("%d\n", a / b);          /* 3   */
```

```
printf("%d\n", a % b);           /* 1   */
printf("%.1f\n", (double)a / b); /* 3.5 */
printf("%zu\n", sizeof(int));    /* 4 on most systems */
```

Check question: What is the value of `m` after the line `double m = 7 / 2;` ?

▶ **Answer**

IN THE COURSE **Chapter 2 · One-Dimensional Arrays in C**

Chapter 13 · Complete Binary Tree in an Array

FN

Functions and Call by Value

parameter · return value · copy

In C all arguments are passed by value: the function receives a copy of the caller's variable. A function that wants to swap two variables must therefore take their addresses, not the variables themselves. In the course, `swap`, the `top` counter of a stack, and insertion into a search tree all take addresses for this reason.

```
void swap_wrong(int x, int y) { int t = x; x = y; y = t; }
void swap(int *x, int *y)     { int t = *x; *x = *y; *y = t; }

int a = 1, b = 2;
swap_wrong(a, b);   /* a == 1, b == 2: only the copies changed */
swap(&a, &b);       /* a == 2, b == 1 */
```

Check question: Although the three assignments inside `swap_wrong` run, why does `a` in `main` not change?

▶ **Answer**

IN THE COURSE **Chapter 1 · SWAP: Macro vs. Function** **Chapter 1 · swap() Step by Step**

Chapter 6 · ++*Top or *Top++?

AS

Arrays and Strings

contiguous memory · char array ending in '\0'

An array is a block in which elements of the same type sit one after another in memory; access `a[i]` takes O(1), and C does no bounds checking. Passing an array to a function means passing the address of its first element; the size must be passed separately. A string is a `char` array that ends with `'\0'`: `"dog"` occupies four bytes.

```
int sum_array(const int a[], int n) {
    int s = 0;
    for (int i = 0; i < n; i++) s += a[i];
    return s;
}
int length(const char *s) {
    int n = 0;
    while (s[n] != '\0') n++;
    return n;
}
/* sum_array((int[]){4, 5, 6}, 3) == 15, length("dog") == 3 */
```

Check question: For the definition `char s[] = "data";`, do `sizeof s` and `length(s)` give the same value?

▶ **Answer**

IN THE COURSE **Chapter 2 · One-Dimensional Arrays in C** **Chapter 2 · String = Char Array** **Chapter 2 · length() Step by Step**

ST

struct and typedef

record · dot and arrow operators · self-referential structure

A `struct` gathers fields of different types into a single record; `typedef` gives the type a short name. A field is reached with `.` through a variable and with `->` through a pointer; writing `p->data` is the same as `(*p).data`. The node of a linked list is a self-referential struct that carries a pointer to its own type.

```
typedef struct node *node_ptr;
typedef struct node {
    int data;
    node_ptr link;       /* next node or NULL */
} node;
```

```
node d = { 3, NULL };
node_ptr p = &d;
p->data = 8;              /* d.data is now 8 */
```

Check question: Can a structure directly contain a field of its own type (`struct node next;`)?

▶ **Answer**

IN THE COURSE Chapter 2 · struct Chapter 2 · Nested and Self-Referential Chapter 4 · The Node Structure

MH

Macros and Header Files

#define · #include · opaque type

The preprocessor substitutes macros defined with `#define` as text before compilation, which is why macro parameters are written inside parentheses. A header file (`.h`) carries the interface of a module and the `.c` file carries its implementation. If the header declares only `typedef struct fraction Fraction;` , the inside of the structure is from the user: this is an opaque type and is the C counterpart of an abstract data type.

```
/* fraction.h */
#ifndef FRACTION_H
#define FRACTION_H
typedef struct fraction Fraction;     /* opaque: fields are not visible
*/
Fraction *fraction_create(int num, int den);
void      fraction_destroy(Fraction *f);
#endif
#define SQUARE(x) ((x) * (x))         /* SQUARE(1 + 2) == 9 */
```

Check question: If the macro were written as `#define SQUARE(x) x * x` , what would `SQUARE(1 + 2)` be?

▶ **Answer**

IN THE COURSE Chapter 1 · Abstraction in C: Header Files Chapter 1 · Using the Interface Chapter 19 · Common Setup: Macros

BE

bool and enum

<stdbool.h> · true/false · named constants

Since C99, the header `<stdbool.h>` provides the `bool` type and the constants `true` and `false`; in the course, insertion and deletion functions report with `bool` whether the operation succeeded. In conditions C still looks at integers: 0 is false and every nonzero value is true. An `enum` gives names to a group of integer constants; the first name gets the value 0 and the following ones increase by one. Operator kinds and node labels can thus be written with readable names instead of numbers.

```
#include <stdbool.h>
typedef enum { white, gray, black } color;   /* 0, 1, 2 */
bool is_empty(int n) { return n == 0; }
color c = gray;
printf("%d %d\n", c, is_empty(0));           /* 1 1 */
```

Check question: In the definition `enum { lparen, rparen, plus, minus, times };`, what value does `times` get?

▶ **Answer**

IN THE COURSE Chapter 8 · get_token Chapter 8 · isp and icp Chapter 9 · Generalized Lists

RC

Recursion and the Call Stack

base case · stack frame · stack overflow

A function that calls itself must make the problem smaller at every call and must reach a base case. For every call, a frame holding the local variables and the return address is added to the call stack; if the depth is n, the extra memory is O(n). Tree traversals, binary search, the towers of Hanoi, and quicksort are written in this style.

```
long fact(int n) {
    if (n <= 1) return 1;        /* base case */
    return n * fact(n - 1);      /* smaller problem */
}
/* fact(4) -> 4 * fact(3) -> 4 * 3 * fact(2)
           -> 4 * 3 * 2 * fact(1) = 24 */
```

Check question: When `fact(4)` is called, what is the largest number of `fact` frames on the call stack at the same time?

▶ **Answer**

IN THE COURSE **Chapter 3 · Recursive Algorithms** **Chapter 3 · Stack Frames** **Chapter 3 · fact(3) Step by Step**

Pointers and memory

PT

Pointers

& and * · NULL · alias

A pointer is a variable that holds the address of an object. `&x` gives the address of x, and `*p` reaches the object that p points to. `NULL` is a pointer that points nowhere and, in the course, represents the empty list and the empty subtree. Two pointers can point to the same object; a change made through one is visible through the other.

```
int x = 5;
int *p = &x;     /* p holds the address of x */
int *q = p;      /* q also points to x       */
*q = 7;          /* x == 7, *p == 7          */
p = NULL;        /* p now points nowhere; x is still 7 */
```

Check question: After the last line, what is the value of the expression `*q` ?

▶ **Answer**

IN THE COURSE **Chapter 4 · Pointer Refresher** **Chapter 4 · Nodes in Memory** **Chapter 6 · Pointers and Memory**

PA

Array and Pointer Arithmetic

a[i] == *(a + i) · counts elements, not bytes

Inside an expression, the name of an array turns into the address of its first element. Adding 1 to a pointer advances it to the next element, not by one byte: if `int` is 4 bytes, the address grows by 4. By definition, `a[i]` means `*(a + i)`. The row-major address computation for multidimensional arrays rests on the same rule.

```
int a[] = {10, 20, 30, 40};
int *p = a;               /* p == &a[0] */
printf("%d\n", *(p + 2)); /* 30 */
```

```
p++;                        /* p == &a[1] */
printf("%d\n", p[1]);       /* 30: p[1] == a[2] */
printf("%td\n", &a[3] - p);  /* 2 elements */
```

Check question: If `int` is 4 bytes and the array `a` starts at address 1000, which address does `a + 3` point to?

▶ **Answer**

IN THE COURSE **Chapter 2 · Arrays and Pointers** **Chapter 2 · print_array() Step by Step** **Chapter 2 · Row Major and Column Major**

DM

Dynamic Memory

malloc · calloc · realloc · free

Local variables disappear when the function ends; a block obtained with `malloc` lives in heap memory until `free` is called. `malloc` returns `NULL` if memory runs out, so its return value is checked every time. A block that is never returned causes a memory leak, and a pointer that is used after the block has been returned is a dangling pointer.

```
int n = 5;
int *a = malloc(n * sizeof *a);
if (a == NULL) return 1;          /* out of memory */
for (int i = 0; i < n; i++) a[i] = i * i;
int *b = realloc(a, 2 * n * sizeof *a);  /* grow */
if (b == NULL) { free(a); return 1; }
a = b;
free(a);
a = NULL;                         /* leave no dangling pointer */
```

Check question: Why does writing `a = realloc(a, new_size);` cause a leak if `realloc` fails?

▶ **Answer**

IN THE COURSE **Chapter 2 · Arrays with malloc** **Chapter 4 · malloc and free** **Chapter 4 · Freeing Step by Step**

PP

Pointer to Pointer

changing the head of a list · &root

If a function is going to change the caller's pointer (for example the head of a list or the root of a tree), it must receive the address of that pointer. The type is then `node **`. This is the call-by-value rule applied to pointers: a pointer is also a variable and is passed by copy.

```
void insert_front(node_ptr *head, node_ptr item) {
    item->link = *head;   /* the new node points to the old head */
    *head = item;         /* the caller's head pointer changes */
}
node_ptr list = NULL;
insert_front(&list, p);   /* list now points to p's node */
```

Check question: Suppose the function were written as `void insert_front(node_ptr head, node_ptr item)` and ended with `head = item;`. If the list is empty, what would `list` be after the call?

▶ **Answer**

IN THE COURSE **Chapter 4 · Inserting at the Front** **Chapter 11 · Why tree_ptr *L**

FP

Function Pointers and qsort

void * · comparison function

The name of a function is also an address and can be passed to another function as an argument. `qsort` in the standard library works without knowing the element type: it takes the array as `void *` and asks the user's comparison function for the order of two elements. The function returns a negative number, zero, or a positive number.

```
int ascending(const void *x, const void *y) {
    int a = *(const int *)x, b = *(const int *)y;
    return (a > b) - (a < b);   /* -1, 0 or 1 */
}
int d[] = {37, 15, 24};
qsort(d, 3, sizeof d[0], ascending);   /* d: 15 24 37 */
```

Check question: In which case does the shortcut `return a - b;` give a wrong result in the comparison?

▶ **Answer**

IN THE COURSE **Chapter 20 · Using qsort** **Chapter 20 · Sorting Records with qsort**

Mathematics

LG

Exponents and Logarithms

lg n = $\log_2$ n · repeated halving

In the course, `lg n` is the logarithm to base 2: it tells how many times n must be halved to reach 1. Since $2^{10} = 1024 \approx 10^3$, lg n ≈ 20 for a million elements. This is why binary search, the height of a balanced tree, and heap operations take O(lg n). The base of the logarithm changes only a constant factor; the base is not written inside big O.

```
int halvings(int n) {
    int k = 0;
    while (n > 1) { n /= 2; k++; }
    return k;                      /* ⌊lg n⌋ */
}
/* 1 → 0, 8 → 3, 1000 → 9, 1024 → 10 */
```

Check question: If a sorted array has 1,000,000 elements, about how many comparisons does binary search make at most?

▶ **Answer**

IN THE COURSE Chapter 3 · Complexity of Binary Search

Chapter 10 · Node Count Relations Chapter 1 · Practical Complexities

Σ

Summation Formulas

arithmetic and geometric series

The cost of nested loops is very often a sum. Two formulas are used constantly in the course: 1 + 2 + … + n = n(n + 1)/2 and $1 + 2 + 4 + \ldots + 2^k = 2^{k+1} - 1$. The first shows that selection sort and insertion sort are $O(n^2)$; the second shows that a binary tree of depth k has at most $2^k - 1$ nodes and that the total number of copies in an array whose capacity doubles stays O(n).

```
int count = 0;
for (int i = 0; i < n - 1; i++)       /* selection sort */
    for (int j = i + 1; j < n; j++)
        count++;                      /* one comparison */
/* count == (n-1) + (n-2) + ... + 1 == n(n-1)/2
   10 for n = 5 */
```

Check question: An array starts with capacity 1 and doubles every time it fills up. If 16 elements are added, how many elements are copied in total during the growths?

▶ **Answer**

IN THE COURSE **Chapter 19 · Selection: Analysis** **Chapter 2 · Amortized Cost** **Chapter 13 · Why O(n)?**

IN

Induction

base step · inductive step

If a statement is true for n = 1 (the base step) and its truth for n − 1 implies its truth for n (the inductive step), then it is true for all n ≥ 1. The correctness of a recursive function is proved the same way: the base case gives the right result, and if the smaller calls are right, the larger call is right too. Lemmas on the number of nodes in trees are proved with this method.

```
/* Claim: total(n) == n * (n + 1) / 2 */
int total(int n) {
    if (n == 0) return 0;            /* base: 0 == 0·1/2 */
    return total(n - 1) + n;         /* (n-1)n/2 + n == n(n+1)/2 */
}
```

Check question: In proving that level i of a binary tree has at most 2^{i-1} nodes, which observation is used in the inductive step?

▶ **Answer**

IN THE COURSE **Chapter 10 · Lemma** **Chapter 3 · Hanoi: How Many Moves?** **Chapter 15 · Theorem 1: Σd = 2e**

MD

Modular Arithmetic

% operator · circular index · hashing

`x % m` is the remainder of dividing x by m and stays between 0 and m − 1 for nonnegative x. In a circular queue, an index that reaches the end of the array wraps around to the start with `(rear + 1) % MAX`; in a hash table, `k % m` maps a key to a bucket. In C the remainder of a negative number can be negative, which is why `unsigned` types are used in hash computations.

```
#define MAX 5
int rear = 4;
```

```
rear = (rear + 1) % MAX;     /* 0: wrapped around to the start */
unsigned h(unsigned k, unsigned m) { return k % m; }
/* m = 13: h(27, 13) == 1, h(40, 13) == 1 → collision */
```

Check question: For m = 13, which buckets do the keys 18 and 44 fall into?

▶ **Answer**

IN THE COURSE **Chapter 7 · Modular Arithmetic** **Chapter 14 · Division Method**

CT

Counting: Factorials and Subsets

n! · 2^n · the limit of brute force

n elements have n! different orderings and 2^n different subsets. Brute-force algorithms try all of these; generating permutations takes O(n · n!), and generating subsets or trying a SAT instance takes O(2^n). Both 2^n and n! grow so fast that even the fastest machine is not enough at about n = 20 for n! and n = 60 for 2^n; this is why backtracking and a good algorithm matter.

```
/* all subsets of n elements: each bit is one element */
for (unsigned m = 0; m < (1u << n); m++) {
    for (int i = 0; i < n; i++)
        if (m & (1u << i)) printf("%d ", a[i]);
    printf("\n");
}   /* 8 lines for n = 3, the first is the empty set */
```

Check question: How many different permutations does the string "abcd" have, and how many letters are printed for each?

▶ **Answer**

IN THE COURSE **Chapter 3 · The Permutation Problem**

Chapter 3 · Generating Subsets Step by Step **Chapter 1 · A Faster Machine Won't Do**

MX

Matrices and Two-Dimensional Arrays

row-major storage · transpose · product

An m × n matrix is held in C as `int a[M][N]` and stored in row-major order: `a[i][j]` is i·N + j elements away from the start of the array. The transpose turns rows into columns: A^T[j][i] = A[i][j]. The product of an m × n matrix and an n × p matrix is m × p; each element is the inner

product of a row and a column, and three nested loops make m·n·p multiplications in total. Sparse matrices, the adjacency matrix of a graph, and all-pairs shortest paths are written with these operations.

```
/* c = a × b : a is M×N, b is N×P, c is M×P */
for (int i = 0; i < M; i++)
    for (int j = 0; j < P; j++) {
        c[i][j] = 0;
        for (int k = 0; k < N; k++)
            c[i][j] += a[i][k] * b[k][j];
    }
```

Check question: A is 2 × 3 and B is 3 × 4. How many elements does AB have, and how many multiplications does computing it take?

▶ **Answer**

IN THE COURSE Chapter 5 · What Is a Transpose?

Chapter 5 · The Idea of Sparse Multiplication

Chapter 15 · Adjacency Matrix

E[X]

Basic Probability and Expected Value

uniform distribution · average case · load factor

Average-case analysis is done under the assumption that the input or the hash values are randomly distributed. The expected value of a quantity is the sum of each value it can take multiplied by its probability; the expected value of a sum is the sum of the expected values of its terms. In sequential search, if the sought element is equally likely to be at any position, the average is (n + 1)/2 comparisons. If n keys are spread uniformly over m buckets, the expected number of keys in a bucket is n/m; this ratio is the load factor α of the hash table.

```
/* if the item is at position k, k comparisons; each position has
probability 1/n */
double avg = 0;
for (int k = 1; k <= n; k++)
    avg += k * (1.0 / n);
/* avg == (n + 1) / 2.0 ; 5 for n = 9 */
```

Check question: 25 keys are uniformly distributed in a chained hash table with 10 buckets. What is the load factor, and how many nodes does an unsuccessful search walk on average?

▶ **Answer**

IN THE COURSE Chapter 14 · Load Factor · Chapter 14 · Chaining or Open Addressing? · Chapter 20 · Average Case (1)

SR

Sets and Relations

union · disjoint sets · equivalence classes

A set is a collection of distinct elements; the union (A ∪ B) gives all elements of the two sets, and the intersection (A ∩ B) gives the common elements. Sets whose intersection is empty are called disjoint. A relation is an equivalence relation if it is reflexive (x ≡ x), symmetric (if x ≡ y then y ≡ x), and transitive (if x ≡ y and y ≡ z then x ≡ z); it divides its elements into disjoint equivalence classes. The union-find structure keeps these classes; Kruskal's algorithm also uses the same structure to ask whether two vertices are in the same component.

```
/* elements 0..4; declarations: 0 ≡ 2, 3 ≡ 4, 2 ≡ 1 */
/* classes: {0, 1, 2} and {3, 4} */
int representative[5] = {0, 0, 0, 3, 3};
bool same_class(int x, int y) {
    return representative[x] == representative[y];
}   /* same_class(1, 0) == true, same_class(1, 4) == false */
```

Check question: Is “being born in the same year” an equivalence relation? What about the relation “x is less than y”?

▶ **Answer**

IN THE COURSE Chapter 16 · Disjoint Sets · Chapter 16 · Two Operations · Chapter 16 · Other Applications

Introduction to algorithm analysis

ST

Counting Steps

program step · number of loop iterations

The time of an algorithm is measured by how many basic steps it runs as a function of the input size n. Every assignment, comparison, and return counts as one step; the condition of a loop is checked once more than the body runs. Constants are details; what matters is how the number of steps grows with n.

```
float sum(float list[], int n) {
    float t = 0;                  /* 1 step      */
    for (int i = 0; i < n; i++)   /* n + 1 steps */
        t += list[i];             /* n steps     */
    return t;                     /* 1 step      */
}                                 /* total 2n + 3 */
```

Check question: A function adds two m × n matrices with two nested loops. Which term does its number of steps grow with?

▶ **Answer**

IN THE COURSE **Chapter 1 · Time Complexity** **Chapter 1 · Step Count: sum()** **Chapter 1 · Step Count: Matrix Addition**

O

Big O, Ω, and Θ

upper bound · lower bound · tight bound

f(n) = O(g(n)) says that there is a positive constant c such that f(n) ≤ c · g(n) for every sufficiently large n: g is an upper bound for f. Ω is the lower bound, and Θ shows that both hold together. The usual order is $1 < \lg n < n < n \lg n < n^2 < n^3 < 2^n < n!$.

```
/* for T(n) = 3n² + 6n + 7:
   for n ≥ 1, 6n ≤ 6n² and 7 ≤ 7n², so
   T(n) ≤ 3n² + 6n² + 7n² = 16n²  →  T(n) = O(n²)
   T(n) ≥ 3n²                     →  T(n) = Ω(n²)
   both together                  →  T(n) = Θ(n²) */
```

Check question: Is it wrong to write $2n + 3 = O(n^2)$?

▶ **Answer**

IN THE COURSE **Chapter 1 · Big-O Notation** **Chapter 1 · Example: $3n^3 + 6n^2 + 7n$** **Chapter 1 · Ω and Θ Notation**

BA

Best, Average, and Worst Case

input-dependent time · amortized cost

Two inputs of the same size may be processed in different amounts of time. In linear search, one comparison suffices if the sought element is first (best case), and n comparisons are needed if it is absent (worst case). The average case requires an assumption about the distribution of inputs. Amortized analysis looks not at single operations but at the total cost of a sequence of operations.

```
int linear_search(const int a[], int n, int x) {
    for (int i = 0; i < n; i++)
        if (a[i] == x) return i;   /* found */
    return -1;                     /* absent: n comparisons */
}
```

Check question: If the sought element is in the array and is equally likely to be at any position, how many comparisons does linear search make on average?

▶ **Answer**

IN THE COURSE **Chapter 1 · Best, Average, Worst** **Chapter 1 · Amortized Analysis** **Chapter 1 · Exercise 2: Sequential Search**

RR

Recurrence Relations

running time of recursive algorithms

The running time of a recursive algorithm is written in terms of itself and solved by expansion. An algorithm that halves the problem at each step and does constant work runs in O(lg n) with T(n) = T(n/2) + c (binary search). An algorithm that solves two halves and combines them with linear work runs in O(n lg n) with T(n) = 2T(n/2) + cn (merge sort). In the towers of Hanoi, the relation h(n) = 2h(n − 1) + 1 gives $2^n - 1$.

```
int binary_search(const int a[], int left, int right, int x) {
    if (left > right) return -1;
    int mid = left + (right - left) / 2;
    if (a[mid] == x) return mid;
    if (a[mid] < x) return binary_search(a, mid + 1, right, x);
    return binary_search(a, left, mid - 1, x);
}   /* T(n) = T(n/2) + c  →  O(lg n) */
```

Check question: Which growth class does the relation T(n) = T(n − 1) + n (with T(1) = 1) belong to?

▶ **Answer**

IN THE COURSE Chapter 3 · Recursive Binary Search

Chapter 3 · Hanoi: How Many Moves? Chapter 20 · Merge Sort: Analysis

SP

Space Complexity and Measurement

extra memory · timing with clock()

Space complexity measures the extra memory needed beyond the input: summing with a loop is O(1), while doing the same work with recursion takes O(n) space on the call stack. Theoretical analysis is also tested with real measurement: `clock()` gives the processor time the program has used. Short tasks are measured by repeating them many times, and slow input and output operations such as `printf` are kept outside the measured region.

```
#include <time.h>
clock_t start = clock();
for (int t = 0; t < 1000; t++)
    linear_search(a, n, -1);        /* worst case */
double sec = (double)(clock() - start) / CLOCKS_PER_SEC;
printf("%g s per search\n", sec / 1000);
```

Check question: If the measured time grows about fourfold when n doubles, what can be said about the growth of the algorithm?

▶ **Answer**

IN THE COURSE Chapter 1 · Space Complexity Chapter 1 · Two Ways to Measure

Chapter 1 · Measurement Results

The code examples are written for C99 and assume that `int` is 4 bytes. The types `node` and `node_ptr` used in the examples are defined in the “struct and typedef” card.

Sercan KÜLCÜ, All rights reserved.

APPENDIX C

Important People in Data Structures and Algorithms with C

Most of the structures and algorithms in this course carry the names of the people who found or popularized them: the AVL tree, the Kruskal and Prim algorithms, Dijkstra's shortest path, Shell sort, Floyd's cycle detection. The books the course builds on also belong to pioneers of the C language and of teaching data structures.

Each card gives the person's years, institution, the contribution that matches something in the course, and links to the slides where that contribution appears. The cards can be filtered by topic, searched, and sorted by surname or by year of contribution.

LE

Leonhard Euler

1707–1783 · St. Petersburg and Berlin academies of sciences

The bridges of Königsberg · 1736

He asked whether a walk exists that crosses each of the seven bridges of Königsberg exactly once. By treating the land masses as vertices and the bridges as edges, he showed that the answer depends only on the connections, not on the shape of the map, and proved that no such walk exists. This 1736 work is regarded as the beginning of graph theory; a closed Euler walk cannot exist unless every vertex has even degree.

IN THE COURSE **Chapter 15 · The 7 Bridges of Königsberg**

Chapter 15 · Path, Connectivity, Degree

graph theory · degree · Euler walk

WH

William George Horner

1786–1837 · Kingswood School; his own school in Bath

Horner's method · 1819

He is known for a method of approximating the roots of polynomials, which had appeared earlier in Ruffini's work. Today his name is attached mostly to the nested way of evaluating a polynomial: written as $a_0 + x(a_1 + x(a_2 + \ldots))$, a polynomial of degree n takes only n multiplications. The course uses the same pattern in two places: finding the relative position of an element in an n-dimensional array, and hashing a string with the loop `h = h * 31 + c`.

IN THE COURSE **Chapter 2 · Generalizing to n Dimensions**

Chapter 2 · offset() Step by Step **Chapter 14 · String Hashing**

polynomial · row-major address · string hash

AC

Arthur Cayley

1821–1895 · University of Cambridge

Number of labeled trees · 1889

In a short note published in 1889 he stated that n labeled vertices can form n^{n-2} different trees. The result first appears in Borchardt's 1860 work, but it is known today as Cayley's formula. The same number counts the spanning trees of the complete graph on n vertices: $4^2 = 16$ for K_4.

IN THE COURSE **Chapter 17 · Definition: Spanning Tree**

Chapter 17 · How Many Spanning Trees?

spanning tree · complete graph · counting

WA

Wilhelm Ackermann

1896–1962 · University of Göttingen (doctorate); secondary schools in Burgsteinfurt and Lüdenscheid

Ackermann function · 1928

This logician, a student of Hilbert, defined a function that is computable but not primitive recursive. The two-variable form used in textbooks today is a simplification by Rózsa Péter and Raphael Robinson. The function grows astonishingly fast; its inverse α(n) grows so slowly that the cost of disjoint-set operations using the weighting rule and the collapsing rule is treated as practically constant.

IN THE COURSE **Chapter 16 · Ackermann and α** **Chapter 16 · Exam Question**

Chapter 16 · Trace: ackermann(1, 2)

recursion · inverse Ackermann · union-find

VJ

Vojtěch Jarník

1897–1970 · Charles University, Prague

Growing a single tree for the minimum spanning tree · 1930

This Czech mathematician, whose main field was number theory, published in Czech in 1930 an algorithm that simplified Borůvka's result: the tree starts from a single vertex, and at each step the cheapest edge joining the tree to the rest of the graph is added. The algorithm was rediscovered by Prim (1957) and Dijkstra (1959), which is why it is also called the DJP algorithm or the Prim–Jarník algorithm.

IN THE COURSE **Chapter 17 · Prim: The Algorithm**

minimum spanning tree · greedy · Prim–Jarník

OB

Otakar Borůvka

1899–1995 · Brno (Masaryk University)

The first minimum spanning tree algorithm · 1926

Prompted by the power-grid problem of an electric company in western Moravia, he published the first known minimum spanning tree algorithm in 1926. In each round every component picks the cheapest edge leaving it; since the number of components at least halves in every round, at most log n rounds suffice. His paper, written in Czech, went unnoticed for a long time; when Sollin rediscovered the method in 1965, the algorithm also became widely known as Sollin's algorithm. Because the rounds are independent of each other, it suits parallel computation.

IN THE COURSE **Chapter 17 · The Greedy Approach**

Chapter 17 · Sollin (Borůvka) and Others

minimum spanning tree · component · parallel

JN

John von Neumann

1903–1957 · Institute for Advanced Study (IAS), Princeton

Merge sort · 1945

In 1945 he wrote a routine for the EDVAC computer that merges two sorted arrays into one sorted array; this routine was part of what is today called merge sort. The manuscript is the oldest surviving example of von Neumann's code. The algorithm is the standard example of the divide-and-conquer approach and of a stable sort that runs in O(n lg n) even in the worst case.

IN THE COURSE **Chapter 20 · Merge Sort** **Chapter 20 · Merge Sort: Analysis**

merging · divide and conquer · EDVAC

HC

H. S. M. Coxeter

1907–2003 · University of Toronto

Magic square rule · Coxeter groups

One of the leading geometers of the century, he worked on higher-dimensional polytopes and reflection groups. He prepared the editions of W. W. Rouse Ball's *Mathematical Recreations and Essays* that appeared after 1939; the textbook bases its magic square rule for odd n on him. The rule finishes in n^2 steps, whereas trying all 25! arrangements by brute force would take millions of centuries. In the course this difference is used to show that a good algorithm matters more than a faster machine.

IN THE COURSE **Chapter 1 · The Magic Square Problem** **Chapter 1 · A Faster Machine Won’t Do**

magic square · brute force · $O(n^2)$

BF

Richard Bellman and Lester R. Ford Jr.

1920–1984 · 1927–2017 · RAND Corporation

Bellman–Ford algorithm · 1956, 1958

Ford in 1956 and Bellman in 1958 published the single-source shortest path method that allows negative edge weights; the same idea also appears in Shimbel (1955) and Moore. All edges are relaxed for n − 1 rounds; if an edge can still be relaxed in round n, the graph contains a negative cycle. Bellman is one of the founders of dynamic programming, and Ford, together with Fulkerson, of maximum-flow theory.

IN THE COURSE **Chapter 18 · The Negative Edge Problem** **Chapter 18 · Bellman-Ford Step by Step**

shortest path · relaxation · negative cycle

RP

Robert C. Prim

1921–2021 · Bell Laboratories

Prim's algorithm · 1957

Working at Bell Laboratories at about the same time as Kruskal, he published in 1957 the algorithm that grows a single tree for the minimum spanning tree, rediscovering Jarník's 1930 method independently. Because it always picks the cheapest edge joining a vertex in the tree to one outside it, no cycle check is needed. The algorithm runs in $O(n^2)$ with arrays and drops to $O(e \log n)$ with a heap, which is faster on sparse graphs.

IN THE COURSE · Chapter 17 · Prim: The Algorithm · Chapter 17 · Prim Step by Step · Chapter 17 · Prim with a Heap (Extra)

minimum spanning tree · greedy · heap

AL

Georgy Adelson-Velsky and Evgenii Landis

1922–2014 · 1921–1997 · Moscow (ITEP, Moscow State University)

AVL tree · 1962

In their 1962 paper “An algorithm for the organization of information” they defined the first self-balancing binary search tree, which keeps the height difference of the two subtrees of every node at most 1; the structure takes its name from the initials of their surnames. When the balance is broken, a single or double rotation restores it, and the height never exceeds $\approx 1.44 \lg n$. Adelson-Velsky was also one of the authors of the Kaissa chess program; Landis's main field was partial differential equations.

IN THE COURSE · Chapter 11 · AVL Definition · Chapter 11 · Single Rotation · Chapter 11 · Double Rotation

AVL tree · balance factor · rotation

AP

Alan J. Perlis

1922–1990 · Carnegie Tech, Yale University

Threaded trees · 1960

One of the designers of ALGOL, he received the first Turing Award in 1966. In 1960, together with C. Thornton, he proposed that the null pointers of a binary tree could be used to point to the inorder predecessor and successor; these pointers are called threads. In a threaded tree, inorder traversal needs neither a stack nor recursion. The short maxims often quoted in the course also come from his “Epigrams on Programming,” published in 1982.

IN THE COURSE **Chapter 12 · Perlis and Thornton** **Chapter 12 · Null Pointers** **Chapter 11 · The Extra Parameter Issue**

thread · inorder traversal · ALGOL

DS

Donald L. Shell

1924–2015 · General Electric; University of Cincinnati (doctorate)

Shell sort · 1959

He published his paper “A High-Speed Sorting Procedure” in Communications of the ACM in 1959. Insertion sort is applied to elements that are h positions apart, and h is reduced down to 1; because distant exchanges remove inversions quickly, the $\Omega(n^2)$ bound of algorithms that exchange only neighbors is broken. The running time depends on the gap sequence, and the sort is not stable because of the distant exchanges.

IN THE COURSE **Chapter 19 · Shell Sort** **Chapter 19 · shell_sort() Step by Step** **Chapter 19 · Number of Inversions**

gap sequence · insertion sort · inversion

JK

Joseph B. Kruskal

1928–2010 · Princeton (doctorate); Bell Laboratories

Kruskal's algorithm · 1956

In 1956 he published the method that finds a minimum spanning tree by sorting the edges by cost and adding to the tree every edge that does not form a cycle. The chosen edges first form a forest, which eventually merges into a single tree. The cycle check is done with disjoint sets: if the two ends of an edge are in different sets, the edge is taken and the sets are merged; the total time is O(e log e). He is also known in statistics for his work on multidimensional scaling.

IN THE COURSE **Chapter 17 · Kruskal: The Algorithm** **Chapter 17 · How to Code Kruskal** **Chapter 16 · Other Applications**

minimum spanning tree · union-find · greedy

ED

Edsger W. Dijkstra

1930–2002 · Mathematisch Centrum Amsterdam; Eindhoven; University of Texas at Austin

Shortest path algorithm · 1959

In a two-page note published in 1959 he gave both the single-source shortest path algorithm and the Jarník–Prim method for the minimum spanning tree. The shortest path algorithm is greedy: the vertex with the smallest distance is finalized, and the distances of its neighbors are updated through it. This choice is correct only for non-negative edge weights. A pioneer of structured programming, Dijkstra received the Turing Award in 1972.

IN THE COURSE **Chapter 18 · The Greedy Idea** · **Chapter 18 · Dijkstra Step by Step** · **Chapter 17 · Prim: The Algorithm**

shortest path · greedy · priority queue

TH

C. A. R. (Tony) Hoare

1934–2026 · Elliott Brothers; Queen's University Belfast; University of Oxford; Microsoft Research

Quicksort · 1961–1962

He devised quicksort in 1959 in Moscow while working on a dictionary translation; he published the algorithm in Communications of the ACM in 1961 and its detailed analysis in The Computer Journal in 1962. Partitioning around a pivot and sorting the two parts recursively gives an in-place sort that runs in O(n lg n) on average. He also developed Hoare logic for program correctness and received the Turing Award in 1980. He died in Cambridge on 5 March 2026.

IN THE COURSE **Chapter 20 · Quicksort** · **Chapter 20 · The Partition Invariant** · **Chapter 20 · Worst and Best Case**

quicksort · partitioning · divide and conquer

SW

Stephen Warshall

1935–2006 · Technical Operations; Massachusetts Computer Associates

Transitive closure · 1962

In 1962 he published the algorithm that computes the transitive closure of a Boolean matrix in $O(n^3)$ time: for each k, the rule “if there is a path from i to k and from k to j, then there is a path from i to j” is applied to all pairs. Floyd carried the same scheme over to path lengths in the same year; the algorithm that finds shortest paths between all pairs is therefore called Floyd–Warshall.

IN THE COURSE **Chapter 18 · Floyd-Warshall Step by Step** **Chapter 18 · Which Algorithm?**

transitive closure · all pairs · $O(n^3)$

RF

Robert W. Floyd

1936–2001 · Carnegie Tech; Stanford University

Floyd–Warshall · cycle detection

In 1962 he published the Floyd–Warshall algorithm, which finds the shortest paths between all pairs of vertices. The “tortoise and hare” method, which finds a cycle in a linked list with two pointers (one advancing one step, the other two), is also attributed to him through Knuth's account; it works with O(1) extra memory. In 1964 he gave the method that builds a heap bottom-up in O(n) time. He received the Turing Award in 1978 for his contributions to program verification.

IN THE COURSE **Chapter 4 · Cycle Detection: Floyd** **Chapter 4 · Floyd Step by Step** **Chapter 18 · Floyd-Warshall Step by Step** **Chapter 13 · Bottom-Up Heap Construction**

tortoise and hare · all pairs · heap construction

DK

Donald E. Knuth

b. 1938 · Stanford University

The Art of Computer Programming · since 1968

With the series *The Art of Computer Programming*, whose first volume appeared in 1968, he made the mathematical analysis of algorithms a discipline; he received the Turing Award in 1974. Many results in the course rest on him: the expected probe counts for linear probing, the

constant $A \approx (\sqrt{5} - 1)/2$ suggested for the multiplication method, the gap sequence 1, 4, 13, 40, … for Shell sort, and the account of Floyd's cycle detection. He is also a co-author of the Knuth–Morris–Pratt string search algorithm. Sedgewick is his doctoral student.

IN THE COURSE **Chapter 14 · Load Factor** **Chapter 14 · Multiplication Method** **Chapter 19 · Shell Sort** **Chapter 4 · Cycle Detection: Floyd**

TAOCP · algorithm analysis · hashing

BM

Rudolf Bayer and Edward M. McCreight

b. 1939 (Bayer) · Boeing Scientific Research Laboratories; TU München (Bayer); Xerox PARC (McCreight)

B-tree · 1970–1972

Working together at Boeing, they designed the B-tree to keep large sorted indexes on disk; the paper was circulated as a report in 1970 and published in Acta Informatica in 1972. In this balanced m-way tree each node holds many keys and all leaves are on the same level; a node that overflows splits in two, sending its middle key up to its parent. The symmetric binary B-trees that Bayer defined in 1972 are the forerunners of red-black trees.

IN THE COURSE **Chapter 11 · Red-Black and B-Tree** **Chapter 11 · Comparing Search Trees**

B-tree · disk · database index

RT

Robert E. Tarjan

b. 1948 · Stanford (doctorate); Princeton University

Analysis of disjoint sets · 1975, 1984

In 1975 he proved that disjoint-set operations using the weighting rule together with path compression are bounded by the inverse Ackermann function α. His 1984 paper with Jan van Leeuwen (b. 1946, Utrecht University) gave a joint worst-case analysis of union and find methods of this kind. With Sleator he defined splay trees, and with Fredman Fibonacci heaps; he received the Turing Award in 1986.

IN THE COURSE **Chapter 16 · Tarjan–van Leeuwen** **Chapter 11 · Balanced Trees** **Chapter 13 · Advanced Heaps**

union-find · splay tree · Fibonacci heap

DR

Dennis M. Ritchie

1941–2011 · Bell Laboratories

The C language · Unix

He developed the C language around 1972 at Bell Laboratories in order to write the Unix operating system; he designed Unix with Ken Thompson and shared the 1983 Turing Award with him. Every structure in the course is built with the tools C provides: `struct` for records, pointers for links, and `malloc` and `free` for run-time memory. *The C Programming Language* (1978), which he wrote with Kernighan, is the constant companion of the course.

IN THE COURSE **Chapter 1 · Textbooks** **Chapter 2 · Arrays and Pointers** **Chapter 2 · struct**

C language · Unix · K&R

BK

Brian W. Kernighan

b. 1942 · Bell Laboratories; Princeton University

The C Programming Language · 1978

The C Programming Language (1978; second edition for ANSI C in 1988), which he wrote with Ritchie, is known as “K&R” after its authors' initials and became the de facto definition of C. With its short, readable examples, the book also set the model for how a programming language book should be written. He is a co-author of the AWK language (the K in its name is his); he has taught at Princeton since 2000.

IN THE COURSE **Chapter 1 · Textbooks** **Chapter 2 · String = Char Array**

K&R · C language · AWK

EH

Ellis Horowitz

b. 1944 · University of Southern California (USC)

Fundamentals of Data Structures · 1976

Fundamentals of Data Structures (1976) and *Fundamentals of Computer Algorithms* (1978), which he wrote with Sahni, were among the most widely used textbooks in the world for many years. The C edition, *Fundamentals of Data Structures in C*, appeared in 1993 with the participation of Anderson-Freed and is the main textbook of this course; references on the slides such as "Horowitz 2.2" point to sections of this book. He chaired the computer science department at USC from 1990 to 1999.

IN THE COURSE **Chapter 1 · Textbooks** **Chapter 1 · Abstract Data Type (ADT)** **Chapter 4 · The Available Space List**

textbook ADT C

SS

Sartaj Sahni

b. 1949 · Cornell (doctorate); University of Florida

The Horowitz–Sahni books

He wrote the data structures and algorithms books together with Horowitz; the textbook's treatment of the system life cycle, the definition of an algorithm, and ADTs comes from this joint work. As a distinguished professor at the University of Florida he worked on parallel computing, NP-completeness, and approximation algorithms. In 1997 he received the IEEE Taylor L. Booth Education Award.

IN THE COURSE **Chapter 1 · Textbooks** **Chapter 1 · System Life Cycle** **Chapter 1 · What Is an Algorithm?**

textbook algorithm parallel computing

SA

Susan Anderson-Freed

d. 2012 · Illinois Wesleyan University

Fundamentals of Data Structures in C · 1993

The third author, who adapted Horowitz and Sahni's book to the C language; *Fundamentals of Data Structures in C*, published in 1993, was translated into five languages. She began as a sociology faculty member at Illinois Wesleyan University and went on to teach programming

courses; in 1986 she founded the university's computer science department and became its first chair. She died on 4 November 2012.

IN THE COURSE Chapter 1 · Textbooks

textbook · C · education

RS

Robert Sedgewick

b. 1946 · Stanford (doctorate); Brown; Princeton University

Analysis of quicksort · Algorithms in C

Under Knuth's supervision he wrote his 1975 doctoral thesis on quicksort; the analysis of improvements such as median-of-three and finishing small partitions with insertion sort comes from this work. *Algorithms in C* is among the supplementary sources of the course; the code in the quicksort chapter is taken from this book. With Guibas he defined red-black trees, and he became the founding chair of the computer science department at Princeton.

IN THE COURSE Chapter 1 · Textbooks · Chapter 20 · Quicksort · Chapter 20 · Median-of-3 and Threshold M

quicksort · Algorithms in C · red-black tree

HW

H. C. von Warnsdorff

Des Rösselsprunges einfachste und allgemeinste Lösung (1823)

Warnsdorff's rule for the knight's tour · 1823

In 1823 he proposed the oldest known heuristic for the knight's tour: at each step the knight moves to the square from which the fewest empty squares can be reached next. This order, which uses up the most cramped squares first, usually finds a complete tour on an 8×8 board without ever backtracking. Solving the same problem with plain backtracking takes far longer. The rule is a classic example of how much work good ordering of candidates saves in backtracking.

IN THE COURSE Chapter 3 · The Knight’s Tour · Chapter 3 · Warnsdorff’s Rule

knight's tour · backtracking · heuristic

JŁ

Jan Łukasiewicz

1878–1956 · Universities of Lviv and Warsaw

Polish (prefix) notation · 1924

A Polish logician. In 1924 he invented prefix notation, which puts the operator before its operands so that logical formulas can be written without parentheses, and he used it in his publications from 1929 on. The notation is therefore called Polish notation; the postfix form, in which the operator is written last, is called reverse Polish notation. A postfix expression is evaluated from left to right with a single stack, needing neither parentheses nor precedence rules.

IN THE COURSE **Chapter 8 · Three Notations** **Chapter 8 · Expression Evaluation** **Chapter 8 · Prefix Evaluation**

prefix notation · postfix notation · expression evaluation

HS

Harold H. Seward

MIT Digital Computer Laboratory (master's thesis, 1954)

Counting sort · 1954

In his 1954 master's thesis at MIT he described counting sort and proposed using it in every pass of radix sort. The algorithm does not compare elements with each other: it first counts how many times each value occurs, then uses the prefix sums of these counts to find each element's place in the output. It sorts n integers in the range 0…k in O(n + k) time and does so stably.

IN THE COURSE **Chapter 19 · counting_sort(): Counting** **Chapter 20 · radixsort(): Units Pass**

counting sort · radix sort · non-comparison sort

JM

John McCarthy

1927–2011 · MIT, Stanford University

LISP · 1958

He designed the LISP language, which appeared in 1958 and was published in *Communications of the ACM* in 1960. In LISP, data are represented by generalized lists whose elements may be atoms or other lists; the head of a list is taken with `car` and its tail with `cdr`. He also invented

garbage collection, which reclaims the memory of list nodes that can no longer be reached, for LISP. He is known as the person who coined the term artificial intelligence and received the Turing Award in 1971.

IN THE COURSE **Chapter 9 · Generalized Lists**

LISP · generalized list · garbage collection

AK

Arthur B. Kahn

Communications of the ACM 5(11), 1962

Topological sort · 1962

In his paper “Topological sorting of large networks” (1962) he gave the algorithm that finds a topological order of a directed acyclic graph by repeatedly removing vertices of in-degree zero. For each removed vertex, the in-degrees of its successors are decreased by one; a vertex whose in-degree drops to zero joins those waiting. When the waiting vertices are kept in a queue, the method is called Kahn's algorithm. It runs in O(n + e) time; if not all vertices can be removed, the graph contains a cycle.

IN THE COURSE **Chapter 18 · Topological Order** **Chapter 18 · Finding a Topological Order** **Chapter 18 · Topological Sort: Analysis**

topological order · in-degree · AOV network

JW

J. W. J. Williams

1930–2012 · Elliott Brothers (London); Bell-Northern Research (Ottawa)

Binary heap and heap sort · 1964

He invented the binary heap and heap sort in 1963 while working at Elliott Brothers, and published the algorithm in 1964 in *Communications of the ACM* as “Algorithm 232: Heapsort.” A heap stores a complete binary tree in an array without gaps; parents and children are found by index arithmetic. Heap sort sorts n elements in O(n lg n) in the worst case without an extra array. The same year Floyd added the method that builds a heap bottom-up in O(n) time.

IN THE COURSE **Chapter 13 · Max Heap and Min Heap** **Chapter 20 · Heap Sort** **Chapter 20 · Heap Sort: Analysis**

binary heap · heap sort · priority queue

JM

Joseph M. Morris

Information Processing Letters 9, 1979

Morris traversal · 1979

In his paper “Traversing binary trees simply and cheaply” (1979) he showed how to traverse a binary tree inorder without a stack, recursion, or a flag field. A temporary thread is written into the empty right pointer of a node's inorder predecessor; the traversal returns along this thread and erases it. The tree is changed temporarily during the traversal and restored at the end. Extra memory is O(1) and time is O(n).

IN THE COURSE **Chapter 12 · Morris Traversal** **Chapter 12 · Morris Step by Step**

Morris traversal · temporary thread · O(1) extra memory

CLRS

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein

Dartmouth College, MIT, Columbia University

Introduction to Algorithms · 1990

The authors of *Introduction to Algorithms*, known as CLRS after their initials. The first edition appeared in 1990 under the names of Cormen, Leiserson, and Rivest; Stein joined the authors in the second edition in 2001. The book is among the supplementary sources of the course: asymptotic notation, recurrence relations, heaps, hash tables, graph algorithms, and the lower bound for comparison sorting are given here with proofs. Rivest shared the 2002 Turing Award with Shamir and Adleman for the RSA cryptosystem.

IN THE COURSE **Chapter 1 · Textbooks**

CLRS · algorithm analysis · textbook

DM

David R. Musser

Rensselaer Polytechnic Institute

Introsort · 1997

He introduced introsort in his 1997 paper “Introspective Sorting and Selection Algorithms.” The algorithm starts with quicksort and tracks the recursion depth; when the depth exceeds about $2\lfloor \lg n \rfloor$, it finishes that part with heap sort. This keeps the average speed of quicksort while

holding the worst case at O(n lg n). The method was developed for the C++ standard library and is common in std::sort implementations. Musser worked with Alexander Stepanov on generic programming and the STL.

IN THE COURSE Chapter 20 · Hybrid Sorts

introsort | quicksort | heap sort

TP

Tim Peters

Python core developer

Timsort · 2002

In 2002 he wrote Timsort for Python. The algorithm finds the already sorted stretches in the data (runs), extends short runs with insertion sort, and merges the runs with merge sort. It is stable, runs in O(n lg n) in the worst case, and in O(n) on already sorted data. It was Python's standard sort from version 2.3 through 3.11; Java SE 7 (for object arrays), Android, and V8 also adopted Timsort.

IN THE COURSE Chapter 20 · Hybrid Sorts

Timsort | stable sort | run

Years and contributions are based on original publications and encyclopedic sources. “b.” marks the year of birth and is used for living people; “d.” marks a person for whom only the year of death is known. Edward McCreight's birth year could not be found in published sources, so it is not given. For people whose life dates could not be found in reliable sources, or who share a combined card, only the year of the contribution and the place of publication are given.

Sercan KÜLCÜ, All rights reserved.