COURSE TEXTBOOK

Data Structures

Sercan Külcü

Giresun University · Department of Computer Engineering

Data Structures

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/`

Slides, study questions and simulators are on that page. The code for each chapter is in the `github.com/sercankulcu/data-structures-java` repository (with Turkish identifiers; the book uses English names).

© Sercan Külcü. All rights reserved.

Generated from the notes of 15 November 2026.

Preface

Data structures determine how data is laid out in memory and how it is accessed; choosing the right structure directly affects how fast operations such as searching, insertion, deletion and traversal run. This book is made up of the notes for the Data Structures course taught in the Department of Computer Engineering at Giresun University, and its chapters follow the order of the course. The examples are written in Java.

The book has four parts. In order, they are **Foundations** (Chapters 1–4), **Linear Structures** (Chapters 5–8), **Trees** (Chapters 9–11), and **Maps, Sets and Graphs** (Chapters 12–14).

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. Code 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 gives, for each English term, the Turkish equivalent used in the Turkish edition of these notes.

The code for each chapter is in its own folder in the companion repository, `github.com/sercankulcu/data-structures-java`; the folder name is given at the start of the chapter. The repository code uses Turkish identifiers, while the listings in this book use English names. Slides, study questions and simulators that run the structures step by step are on the course page.

Sercan Külcü
Giresun

Contents

Course Map 1

PART I · FOUNDATIONS

1 Introduction 4

1.1 Digital data 4
1.2 Data structures and algorithms 10
1.3 Data types 12
1.4 The Java memory model 15
1.5 Generics 21
1.6 Non-primitive data structures 25
1.7 Abstract data types and data structures 27
1.8 Organizing data and choosing a structure 35
1.9 The fundamental data structures at a glance 38
1.10 Common mistakes 40
1.11 Summary 40
1.12 Key terms 41
1.13 Test Yourself 42
1.14 Further reading 44
1.15 References 44

2 Algorithm Analysis 45

2.1 Algorithms and measurement 45
2.2 Asymptotic notation 54
2.3 From code to complexity 64
2.4 Space, correctness and robustness 80
2.5 Common mistakes 85
2.6 Summary 86
2.7 Key terms 87
2.8 Test Yourself 88
2.9 Further reading 90
2.10 References 90

3 Arrays 91

3.1 The idea of an array 92
3.2 Arrays in Java 97
3.3 Array algorithms 109
3.4 Multidimensional arrays 121
3.5 The complexity of array operations 137
3.6 Common mistakes 151
3.7 Summary 151
3.8 Key terms 153

3.9 Test Yourself 153
3.10 Further reading 155
3.11 References 156

4 Recursion 157

4.1 What is recursion? 157
4.2 The call stack 164
4.3 Linear recursion 170
4.4 Binary recursion 177
4.5 The Towers of Hanoi and backtracking 185
4.6 Recurrence relations 194
4.7 Tail recursion 199
4.8 Recursion on linked lists 203
4.9 Common mistakes 207
4.10 Summary 207
4.11 Key terms 209
4.12 Test Yourself 209
4.13 Further reading 211
4.14 References 211

PART II · LINEAR STRUCTURES

5 Linked Lists 213

5.1 Fundamentals 214
5.2 Singly linked lists: traversal and insertion 222
5.3 Singly linked lists: deletion and other operations 232
5.4 Doubly linked lists 247
5.5 Circularly linked lists 262
5.6 Array or linked list? 272
5.7 Common mistakes 274
5.8 Summary 274
5.9 Key terms 275
5.10 Test Yourself 276
5.11 Further reading 278
5.12 References 278

6 The List ADT and Java Collections 279

6.1 Collection interfaces 280
6.2 The list ADT 286
6.3 List methods 292
6.4 Traversal 303
6.5 ArrayList 314
6.6 LinkedList and the positional list 322
6.7 Other list classes 330
6.8 Summary 337
6.9 Key terms 338

6.10 Test Yourself 338
6.11 Further reading 340
6.12 References 341

7 Stacks 342

7.1 The stack ADT 343
7.2 Implementation 349
7.3 Stack applications 361
7.4 Towers of Hanoi and stock span 374
7.5 Expression notations 380
7.6 Common mistakes and a common pattern 392
7.7 Summary 392
7.8 Key terms 394
7.9 Test Yourself 394
7.10 Further reading 396
7.11 References 396

8 Queues and Deques 397

8.1 The queue ADT 397
8.2 Array-based queues 404
8.3 Linked queues 419
8.4 Kinds of queues and the deque 425
8.5 Applications of queues 436
8.6 Common mistakes 446
8.7 Summary 446
8.8 Key terms 447
8.9 Test Yourself 448
8.10 Further reading 450
8.11 References 450

PART III · TREES

9 Trees 453

9.1 The tree concept 454
9.2 Binary trees 468
9.3 Representations 477
9.4 Tree traversals 485
9.5 Coding the traversals 493
9.6 Applications of traversals 505
9.7 Common mistakes 515
9.8 Summary 516
9.9 Key terms 517
9.10 Test Yourself 517
9.11 Further reading 519
9.12 References 520

10 Search Trees 521

10.1 Search trees 521
10.2 Binary search trees 524
10.3 BST operations 532
10.4 Balance and AVL trees 555
10.5 Red-black trees 571
10.6 Splay trees 581
10.7 Multiway search trees 586
10.8 Common mistakes 592
10.9 Summary 592
10.10 Key terms 593
10.11 Test Yourself 594
10.12 Further reading 596
10.13 References 596

11 Priority Queues and Heaps 597

11.1 The priority queue ADT 597
11.2 List-based implementations 602
11.3 Binary heaps 611
11.4 Insertion and removal 618
11.5 Building a heap and heap sort 633
11.6 Java and applications 641
11.7 Common mistakes 646
11.8 Summary 647
11.9 Key terms 648
11.10 Test Yourself 648
11.11 Further reading 650
11.12 References 651

PART IV · MAPS, SETS AND GRAPHS

12 Maps and Hash Tables 653

12.1 Maps and the idea of hashing 654
12.2 Hash functions 659
12.3 Separate chaining 670
12.4 Open addressing 683
12.5 Performance and applications 693
12.6 Common mistakes 705
12.7 Summary 706
12.8 Key terms 707
12.9 Test Yourself 707
12.10 Further reading 709
12.11 References 709

13 Sets and Disjoint Sets 710

13.1 The set ADT 711
13.2 Implementing sets 716

13.3 Multisets and multimaps 729
13.4 Disjoint sets and Union-Find 734
13.5 Making Union-Find fast 743
13.6 Common mistakes 758
13.7 Summary 759
13.8 Key terms 760
13.9 Test Yourself 760
13.10 Further reading 762
13.11 References 763

14 Graphs 764

14.1 Basic concepts 765
14.2 Graph representations 775
14.3 Traversal: BFS and DFS 786
14.4 Shortest paths 806
14.5 Minimum spanning trees 819
14.6 Summary 830
14.7 Key terms 832
14.8 Test Yourself 832
14.9 Further reading 834
14.10 References 834

APPENDICES

Appendix A Glossary of Terms 836

Appendix B Prerequisites for Data Structures 841

Appendix C Important People in Data Structures 847

DATA STRUCTURES

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.

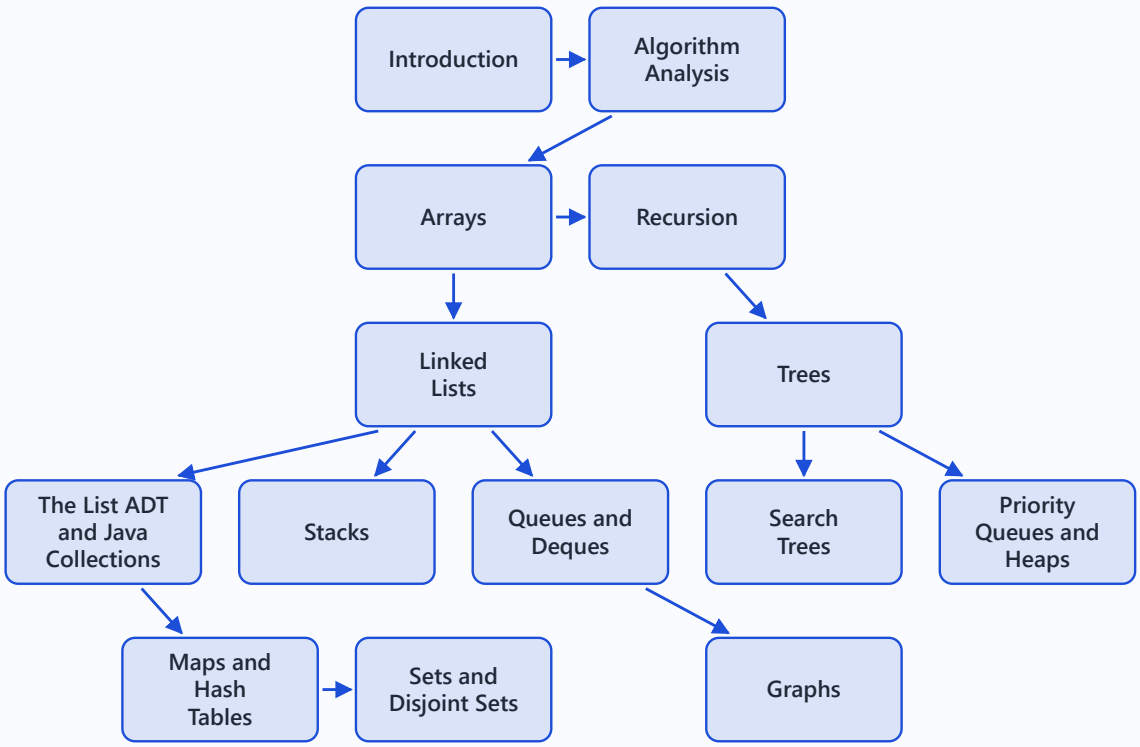


Figure 1: The course map. Each arrow points from a topic to one that builds on it. Starting from arrays, the path branches into linked structures, trees, and graphs.

Figure 1 draws only the most direct connections. Table 1 lists every chapter that each chapter depends on, together with the idea that links them.

Topic (chapter)	Builds on	Connection
Introduction (1)	—	How data is represented in memory, primitive and reference types, abstract data types: the vocabulary in which every structure is described.
Algorithm Analysis (2)	Introduction	Choosing a structure means measuring what its operations cost: Big O, Omega, and Theta.

Topic (chapter)	Builds on	Connection
Arrays (3)	Algorithm Analysis	The first structure: contiguous memory with O(1) access by index; analysis shows what insertion and deletion cost.
Recursion (4)	Algorithm Analysis, Arrays	Reducing a problem to smaller instances of itself, analyzed through the call stack and recurrence relations.
Linked Lists (5)	Arrays	Nodes remove the fixed size and shifting cost of arrays: O(1) insertion at the front, but no random access.
The List ADT and Java Collections (6)	Arrays, Linked Lists	Arrays and linked lists are two implementations of the same List ADT; Java collections and Iterator.
Stacks (7)	Arrays, Linked Lists	A list accessed only at the top: push and pop are O(1) with either an array or a linked list.
Queues and Deques (8)	Arrays, Linked Lists, Stacks	Access at both ends: a circular array or a linked list; a queue built from two stacks.
Trees (9)	Recursion, Linked Lists	A node can now link to several children; traversals are written recursively.
Search Trees (10)	Trees	An ordering condition makes search, insertion, and deletion O(h); balancing keeps the height O(log n).
Priority Queues and Heaps (11)	Queues and Deques, Trees	A priority version of the Queue ADT; a heap stores a complete binary tree in an array.
Maps and Hash Tables (12)	Arrays, The List ADT and Java Collections	The Map ADT; a hash table of array buckets and chaining lists gives O(1) on average.
Sets and Disjoint Sets (13)	Search Trees, Maps and Hash Tables	A set is a map without values: hash-based and tree-based implementations; disjoint sets as trees.
Graphs (14)	Queues and Deques, Trees, Priority Queues and Heaps, Sets and Disjoint Sets	A generalization of trees: BFS uses a queue, Dijkstra and Prim a priority queue, Kruskal disjoint sets.

Table 1: Connections on the map.

The backbone of the map is **analysis → array → linked structure → tree → graph**. The first four chapters supply the tools for the whole course: every structure is defined as an abstract data type, and its operations are measured with Big O. Arrays and linked lists are the two basic ways of implementing everything that follows; stacks, queues, and the List ADT are built on top of them. Recursion leads to trees, and search trees and hash tables are the two ways of implementing maps and sets. The final chapter, on graphs, uses all of it: queues and stacks for traversal, priority queues, and disjoint sets.

PART I

Foundations

Chapters 1–4

PART I · CHAPTER 1

Introduction

Every program ends up working on data in memory, and how that data is represented, organized, and operated on decides both whether the program is correct and how fast it runs. We start with text, numbers, images, and sound as 0s and 1s, then turn to Java: primitive types, type safety, the call stack and the heap, and copying a value versus copying a reference. Generics let one class serve every reference type. At the center of the chapter is the split between an abstract data type and a data structure — an interface says what, a class says how — made concrete by a Java stack traced line by line. The closing sections cover ways of organizing data, how to choose a structure, and the fundamental structures of the course.

Code for this chapter: `github.com/sercankulcu/data-structures-java/Bolum01/src`

LEARNING OUTCOMES

Core concepts
Tell a data type, an abstract data type, and a data structure apart.

Choosing a structure
Choose a data structure from a problem's requirements.

ADTs in Java
Write an ADT as an interface and a data structure as a class; use references and generics correctly.

Fundamental structures
Know the fundamental structures, how they organize data, and their typical uses.

1.1 Digital data

This course is about how to organize data in memory, so it makes sense to start with the data itself. A computer's memory holds no letters, photographs, or songs. It holds billions of tiny cells, each set to 0 or 1, and everything else is built on top of them. How does a song end up as a pattern of bits?

1.1.1 What is digital data?

When you record a voice memo, your voice reaches the phone as a pressure wave in the air. The wave is *continuous*: it has a value at every instant, and the value changes smoothly from one instant to the next. Quantities like this are called **analog**. The microphone converts the wave into a voltage, which is still analog. The processor, however, can only work with numbers — strings of 0s and 1s.

An **analog-to-digital converter (ADC)** bridges the gap. It samples the voltage at regular intervals and records each sample as a number. Playback reverses the process: a **digital-to-analog converter (DAC)** turns the numbers back into a voltage, and the speaker turns the voltage back into sound. Everything in between — saving, copying, sending, changing the volume, filtering out noise — is arithmetic on numbers (Figure 1.1).



Figure 1.1: Analog sound → ADC → 0s and 1s → computer → DAC → analog sound. The computer only ever sees the numbers in the middle.

Definition 1.1

Digital data is information — numbers, text, images, sound — represented as sequences of 0s and 1s. Text documents, image files, databases, and audio recordings are all digital data.

The important word here is *represented*. A pattern of bits means nothing by itself: the same eight bits might be a letter in one file, a number in another, and the red level of a pixel in a third. What gives bits meaning is an **encoding**, a rule for interpreting them. A file extension such as .txt, .png, or .mp3 is a hint about which encoding to use.

Common mistake

People often use “digital” to mean “on a computer.” The real distinction is between values that vary continuously and values that come in discrete steps. The hand of an analog clock can point anywhere; a digital clock jumps from 14:07 to 14:08 and shows nothing in between. Going digital always loses some information. The engineering problem is to keep the loss too small to notice.

1.1.2 Bits, bytes, and binary numbers

The smallest unit of memory is the **bit** (*binary digit*), which is either 0 or 1. In hardware, the two values might be a low or high voltage, or a tiny patch of disk magnetized one way or the other. Two states are much easier to tell apart reliably than ten, which is the main reason computers work in binary.

One bit carries little information, so bits are grouped; eight bits make a **byte**. Every extra bit doubles the number of patterns, since each existing pattern can be followed by a 0 or a 1. One bit gives 2 patterns, two bits give 4 (00, 01, 10, 11), and n bits give 2^n: 256 for a byte, 65,536 for 16 bits, and 4,294,967,296 for 32.

To read a bit pattern as a number we use the **binary number system**. In decimal, digits are weighted by powers of ten — 365 = 3·100 + 6·10 + 5·1. Binary works the same way with powers of two, so the eight positions of a byte are worth 128, 64, 32, 16, 8, 4, 2, and 1.

Example 1.1

Write 13 as a byte, then find the value of 01000001.

1. **Decompose 13:** The largest power of two not exceeding 13 is 8, leaving 5; then 4, leaving 1; then 1. So 13 = 8 + 4 + 1.
2. **Fill in the positions:** Put 1s under 8, 4, and 1 and 0s everywhere else: `00001101`.
3. **The other direction:** In `01000001` the 1s sit under 64 and 1, so the value is 65.
4. **Check:** `00001011` has 1s under 8, 2, and 1 — that is 11, not 13. Shifting a single bit gives a completely different number.

Larger units come up all the time. A kilobyte (KB) is about a thousand bytes, a megabyte (MB) about a million, and a gigabyte (GB) about a billion. "About," because some software counts in powers of 1,024 (2^{10}, a round number in binary) rather than 1,000.

Common mistake

Bits and bytes are easy to confuse. Network speeds are quoted in bits per second (Mbit/s), file sizes in bytes (MB). A 100 Mbit/s connection moves at most 100 / 8 = 12.5 MB per second, not 100 MB.

1.1.3 Why digital?

Three properties explain why digital data took over.

- **It is numeric.** Everything — photos, novels, songs — becomes binary numbers such as `1011 0010`, so the same hardware can store and carry all of it. A hard disk neither knows nor cares what its bits mean.
- **It is easy to store and share.** Data moves freely between disks, the cloud, and networks, and copying is lossless. A cassette degrades a little with every copy; the millionth copy of a digital file is identical, bit for bit, to the first.
- **It is fast to process.** An ordinary processor today performs on the order of 10^9 — a billion — basic operations per second.

The third point is also the reason this course exists. A billion operations a second sounds like enough for anything, but a poorly organized program can do work that grows far faster than its input. Suppose a program compares each of a million records with every other record. That is $10^6 \cdot 10^6 = 10^{12}$ comparisons, or about 1,000 seconds — more than sixteen minutes. Keep the same records in a suitable structure and the job can finish in a tiny fraction of a second. The machine sees only bits; data structures are how we arrange those bits so that the work stays small.

1.1.4 Representing text, numbers, images, and sound

Each kind of information has its own encoding. Table 1.1 summarizes the four basic cases, and the paragraphs that follow look at each in turn.

Kind	Represented as	Example
Text	character encodings such as ASCII and Unicode	`'A'` → 65 → `01000001`
Numbers	integers, floating-point numbers	13 → `00001101` · 3.14 → IEEE 754
Images	color values of pixels	red pixel → (R, G, B) = (255, 0, 0)
Sound	digital samples of the sound wave	CD quality → 44,100 samples per second

Table 1.1: How digital data is represented.

Text. To store text, each character is assigned a number by a **character encoding**. One of the oldest and most widespread is **ASCII** (American Standard Code for Information Interchange), which numbers 128 characters from 0 to 127: uppercase letters start at 65 (`'A'`), lowercase at 97 (`'a'`), and digits at 48 (`'0'`). Seven bits are enough, though in practice each character takes a byte. `'A'` is stored as 65, that is, `01000001`.

ASCII covers only English letters; Turkish ş, ğ, and ı are missing, as are the scripts of most of the world. **Unicode** fixes this by giving every character in every writing system a *code point* — ş, for instance, is 351. How a code point is laid out in bytes is a separate rule. UTF-8, the most common choice on the web, stores ASCII characters in one byte and others in two or more, so a word can take more bytes than it has letters. The Turkish word *veri* ("data") is 4 letters and 4 bytes; *şeker* ("sugar") is 5 letters but 6 bytes, because ş takes two. Java's `char`, by contrast, is a 16-bit UTF-16 code unit, so ş fits in a single `char`.

Example 1.2

How is the word “Veri” stored in ASCII?

1. **Codes:** `'V'` = 86, `'e'` = 101, `'r'` = 114, `'i'` = 105.
2. **In binary:** 86 = 64 + 16 + 4 + 2 → `01010110` ; 101 = 64 + 32 + 4 + 1 → `01100101` ; 114 = 64 + 32 + 16 + 2 → `01110010` ; 105 = 64 + 32 + 8 + 1 → `01101001` .
3. **Result:** four consecutive bytes, `01010110 01100101 01110010 01101001` . Nothing in these 32 bits says “word”; read as an `int` , they are simply a large integer. The meaning comes from the encoding the reading program applies.

Numbers. Integers are written directly in binary. For negative numbers, computers use a scheme called *two's complement*. We don't need its details here, only the result: an n-bit signed integer ranges from -2^{n-1} to $2^{n-1} - 1$, so a byte holds −128 to 127. Numbers with a fractional part are stored in **floating-point** form, defined by the IEEE 754 standard. Like scientific notation, a floating-point number has a sign, a significand, and an exponent; 3.14 is stored as roughly $1.57 \cdot 2^1$. Since the significand has a fixed number of digits, most decimal fractions can only be stored approximately.

Images. A digital image is a grid of tiny squares called **pixels**. Each pixel's color is given by three numbers saying how much red (R), green (G), and blue (B) light to mix, usually one byte each, from 0 to 255. Pure red is (255, 0, 0), white is (255, 255, 255), and black is (0, 0, 0). The image itself is a long sequence of these triples, row after row.

Sound. Sound becomes numbers when the ADC samples it. Each measurement is a **sample**, and the number of samples per second is the **sampling rate**. An audio CD takes 44,100 samples per second, each a 16-bit integer, with separate samples for the left and right channels.

Example 1.3

How much space does one minute of uncompressed CD audio take? And an uncompressed 1920 × 1080 photo?

1. **Audio, one second:** 44,100 samples × 2 bytes × 2 channels = 176,400 bytes.
2. **Audio, one minute:** 176,400 × 60 = 10,584,000 bytes, about 10.6 MB.
3. **Photo:** 1920 × 1080 = 2,073,600 pixels at 3 bytes each = 6,220,800 bytes, about 6.2 MB.
4. **Takeaway:** The MP3 and JPEG files on a phone are far smaller, because these formats discard detail the ear and eye won't miss. Compression is itself a way of organizing data: it looks for ways to say the same thing in fewer bits.

Common mistake

A digit character is not the number it shows. The character `'7'` has ASCII code 55, not 7. Text read from a file must be converted before you can do arithmetic with it. In Java, `'7' - '0'` gives 55 − 48 = 7, which is the quick way to turn a digit character into its value.

1.1.5 Kinds of digital data

Digital data is also classified by how it is organized. Table 1.2 shows a common classification.

Kind	Description	Examples
Structured	organized according to a fixed schema	relational database tables, CSV
Semi-structured	partly organized by tags	XML, JSON
Text	human-readable text	TXT, email
Images and graphics	visual data	JPG, PNG
Audio and music	sound recordings	MP3, WAV

Table 1.2: Kinds of digital data.

In **structured** data, the fields of every record are fixed in advance. Each row of a student table has an ID, a name, a department, and a GPA; this list of fields is the table's *schema*. A CSV (comma-separated values) file is structured too: each line is a record and each comma-separated column a field. Because every value has a known place, structured data is easy to search and sort.

Semi-structured data has no schema, or only a loose one; instead, tags describe the data as it goes. Here is a student in JSON (JavaScript Object Notation):

```
{ "id": 101, "name": "Alice", "courses": ["Data Structures", "Physics"] }
```

Each value is labeled (`"id"`, `"name"`). Another student's record might have no `"courses"` field, or an extra `"phone"` field. XML (eXtensible Markup Language) does the same with opening and closing tags, as in `<name>Alice</name>`. Semi-structured data nests naturally — a student holds a list of courses, each course a list of grades — and nesting of this kind is a tree.

Text, images, and sound are organized by content rather than by a schema. The body of an email or the pixels of a photo don't divide into fields the way a table does, which is why such data is often called *unstructured*.

The lesson for data structures is that the outside shape of the data shapes the structure we choose inside the program. CSV rows fit naturally in an array or list; the nesting of JSON and XML calls for a tree; and finding a person from an email address calls for a key–value map.

1.2 Data structures and algorithms

A program consists of data and the steps applied to it, and the two are not independent. How the data is stored decides which steps are easy and which are laborious; the steps needed decide how the data should be stored.

1.2.1 Algorithms and data structures

Definition 1.2

An **algorithm** is a step-by-step plan for carrying out a task: a finite, ordered set of unambiguous instructions. It is one of the basic building blocks of computer science.

A recipe is the everyday example: listed ingredients, ordered steps that each say exactly what to do, and an end. A computer algorithm needs the same, with steps precise enough for a machine — no "add a pinch of salt" — and a finite number of them.

Definition 1.3

A **data structure** is a way of organizing and storing data in a computer. It specifies how the data is laid out in memory and which operations reach it.

The data structure is the groundwork and the algorithm runs on top; choose both well and the software is efficient and the problem solved systematically. Niklaus Wirth, the Swiss computer scientist, put the relationship in the title of his 1976 book: *Algorithms + Data Structures = Programs*.

Suppose we look up a name in a thousand-entry phone book. Unordered, the names must be checked one by one, and a name that is last or missing costs a thousand comparisons. Alphabetized, we can check the middle name and discard the right half if ours comes before it, the left half if after. Each comparison halves the candidates — 1000, 500, 250, 125, 63, 32, 16, 8, 4, 2, 1 — so 10 comparisons at most settle it. Kept in a different order, the same data allowed an algorithm a hundred times faster. The price: every new name must go in its proper place.

Common mistake

A data structure is not just where data sits, and the algorithm is not a separate topic. The halving method above gives wrong answers on an unsorted list. Whether an algorithm is correct and fast depends on the guarantees of its data structure.

1.2.2 What organization costs and what it buys

Data structures exist for fast, effective access. The slides picture two kitchen drawers with the same utensils, one jumbled, one divided into compartments. Finding a spoon in the first may mean handling every item; in the second, your hand goes straight to the spoons.

The subtler lesson: organization is not free. The tidy drawer makes you find the right compartment; the jumbled one takes things anywhere. Organization makes insertion a little harder and search much easier, so the choice depends on which happens more often. If one item goes in each year and something is looked for daily, organizing pays. If things go in daily and are never looked for — a wastebasket — it is wasted effort.

This **trade-off** is the heart of choosing a data structure. Every structure makes some operations cheap and others expensive, and that choice sets a program's speed and memory use.

1.2.3 Building blocks: objects, arrays, links

However complex they look, nearly all data structures are built from three parts.

Objects. An object gathers related attributes into one unit with named fields. A student's name and ID could live in two variables, but then the programmer must remember which name goes with which number. In Java, a class bundling the two looks like this:

```
class Student {
  String name; // the student's name
  int id;      // student ID number
  Student(String name, int id) { this.name = name; this.id = id; }
}
```

A `Student` object keeps its `name` and `id` together in memory, and they travel together. Objects are also the basis of *nodes*: a node is a small object holding a value plus fields pointing to its neighbors.

Arrays. An array stores elements of one type in consecutive memory slots, and that contiguity is its great strength. Knowing where the array starts and how many bytes each element takes, we find element i with one multiplication and one addition: start address + i × element size. For 4-byte `int`s starting at address 1000, element 5 is at 1000 + 5 × 4 = 1020. The cost does not depend on length: ten elements or ten million, it is the same.

Links. A link points from one entity to another; Java calls it a **reference**. Links set up optional, directed relationships: a student points to an advisor, a city to its neighbors, a song to the next song in a playlist. A linked structure's elements can be scattered through memory, kept in order by arrows rather than by contiguity. In the slides' figure the last arrow ends in a question mark: a chain must end somewhere, and Java writes that final link to nothing as `null`.

Arrays versus links is one of the course's main themes. An array gives instant access by index, but its size is fixed and inserting in the middle shifts elements. A linked structure grows freely and inserts by redirecting a few arrows, but reaching element *i* means following the arrows from the front.

1.3 Data types

A language learns from the programmer how to interpret bits: declaring a variable's type says “read these bits as an integer” or “as a character.” Types also guard a program against errors.

1.3.1 Primitive types

Java has eight basic types built into the language, the **primitive types**, for integers, floating-point numbers, logical values, and characters. Table 1.3 lists their sizes and ranges.

Java type	Size	Values
`byte`	8 bits	−128 … 127
`short`	16 bits	−32,768 … 32,767
`int`	32 bits	about ±2.1 billion (−2,147,483,648 … 2,147,483,647)
`long`	64 bits	about $\pm 9.2 \cdot 10^{18}$
`float`	32 bits	floating-point (≈7 significant digits)
`double`	64 bits	floating-point (≈15 significant digits)
`boolean`	—	`true` / `false`
`char`	16 bits	UTF-16 code unit

Table 1.3: Java's primitive types.

The ranges follow from the rule in 1.1.4: an n-bit signed integer runs from -2^{n-1} to $2^{n-1} - 1$. For `byte`, $n = 8$ and $2^7 = 128$, so −128 … 127. For `short`, $2^{15} = 32{,}768$; for `int`, $2^{31} = 2{,}147{,}483{,}648$, hence “about ±2.1 billion”; for `long`, $2^{63} \approx 9.22 \cdot 10^{18}$. The positive side is one short because zero uses up a pattern in the non-negative half.

Primitive types share four properties. They are simple, not built from other types. They have a fixed size: an `int` takes 32 bits whether it holds 3 or 2 billion. The processor handles them directly; adding two `int`s is one instruction. And a primitive variable holds its value inside itself, which will matter in 1.4.

A fixed size means a limit. Past it, Java reports no error; the value *wraps around* to the other end of the range. This is **overflow**.

```
int big = Integer.MAX_VALUE;            // 2147483647
System.out.println(big + 1);            // -2147483648
long wide = (long) big + 1;
System.out.println(wide);               // 2147483648
System.out.println(0.1 + 0.2);          // 0.30000000000000004
```

Example 1.4

What happens in the first two lines of the code above?

1. **Starting value:** `big` holds 2,147,483,647, the largest `int`: in binary, a 0 followed by thirty-one 1s.
2. **Adding one:** The carry runs through every position, leaving a 1 followed by thirty-one 0s.
3. **Reading the result:** In a signed integer a leading 1 means negative. This pattern is the smallest `int`, −2,147,483,648, which the program prints for `big + 1`.
4. **Remedy:** Use `long` where a result may exceed the `int` range. Methods such as `Math.addExact` throw an exception on overflow instead of wrapping silently.

Floating-point types are limited differently: they reach very large and very small numbers, but only approximately. A `float` keeps about 7 significant digits, a `double` about 15. In binary, 0.1 is an infinite fraction (as 1/3 is 0.333… in decimal), so memory holds an approximation. Hence the last line: `0.1 + 0.2` prints `0.30000000000000004`.

Common mistake

Floating-point numbers should not be compared with `==`: `0.1 + 0.2 == 0.3` is `false`. Two `double`s count as equal when the absolute value of their difference is below a small bound such as 10^{-9}. Values that must be exact to the cent, such as money, belong in a `long` counting cents or a `BigDecimal`, not a `double`.

1.3.2 A type is values plus operations

A type is more than "the values it can take." `int` and `String` differ not only in holding numbers versus text but in the operations that apply to them.

Definition 1.4

A **data type** defines a set of values together with their behavior. The **set of values** is the type's valid values; the **behavior** is the operations that can be performed on them.

Integers support addition, subtraction, multiplication, and division. Strings (`String`) support comparison, concatenation, and substrings: `"data" + "base"` gives `"database"`, and `"database".substring(0, 4)` gives `"data"`. The same `+` adds numbers and joins strings — clear evidence that behavior belongs to the type.

Division shows it best. In Java, an `int` divided by an `int` is an `int`; the fraction is not rounded but dropped.

```
System.out.println(7 / 2);     // 3
System.out.println(7 % 2);     // 1
System.out.println(-7 / 2);    // -3
System.out.println(-7 % 2);    // -1
System.out.println(7.0 / 2);   // 3.5
System.out.println(7 / 2.0);   // 3.5
int n1 = 7, n2 = 8;
double avg1 = (n1 + n2) / 2;     // 7.0
double avg2 = (n1 + n2) / 2.0;   // 7.5
```

`7 / 2` drops the fraction of 3.5 and gives 3. `7 % 2` is the remainder, 1, so that 7 = 2 · 3 + 1. `-7 / 2` is −3, not −4, because Java truncates toward zero, and the remainder matches: −7 = 2 · (−3) + (−1). With a `double` operand, as in `7.0 / 2` or `7 / 2.0`, the division is in floating point and gives 3.5.

Example 1.5

The last two lines average 7 and 8 in two ways. Why do the results differ?

1. **First line:** `(n1 + n2)` is the `int` 15, and `15 / 2` is integer division: 7. Only then is 7 converted to `double`, so `avg1` gets 7.0. The loss came before the assignment.
2. **Second line:** `2.0` is a `double`, and Java promotes the other operand to match: `15.0 / 2.0 = 7.5`.
3. **Lesson:** The type of the receiving variable does not decide how the operation is done; the operand types do.

Mixed types follow fixed rules. A `char` is really a 16-bit number and becomes an `int` in arithmetic, so `'A' + 1` is 66. If one side of `+` is a `String`, the other is converted to text and appended: `"A" + 1` gives `"A1"`. Evaluation runs left to right, so `1 + 2 + "x"` gives `"3x"` but `"x" + 1 + 2` gives `"x12"`.

Common mistake

Integer division is easy to forget in averages, percentages, and ratios. If 7 of 30 students passed, `7 / 30 * 100` gives 0, because `7 / 30` is 0. Either multiply first, so that `7 * 100 /`

`30` gives 23, or make one operand floating-point, so that `7.0 / 30 * 100` gives about 23.33.

1.3.3 Type safety

Not every operation fits every type. Multiplying text by a number means nothing in Java: for `"abc" * 2` the compiler reports an error without running the program. This protective role is called **type safety**. Types keep errors to a minimum; their restrictions tell the programmer what is possible.

When an error is caught matters greatly. In a *statically typed* language like Java, types are written in the source, and the compiler checks every operation against them before the program runs: **compile-time** checking. Such an error is found and fixed before anyone else sees the program. Had it surfaced only at **run time**, the program might have worked for months and then crashed in a user's hands the first time it reached that line.

Type safety is especially valuable in data structures. A structure holds thousands of elements, added and removed all over a program, and one element of the wrong type causes trouble far from where it entered. So Java data structures are written with generics, which tell the compiler their element type (1.5).

Common mistake

A program that compiles is not necessarily correct, since type safety catches only type violations. `7 / 30 * 100` is well typed and compiles; the programmer's intent is what is wrong. The compiler catches type errors, not logic errors.

1.4 The Java memory model

Nearly every data structure rests on a variable “pointing to” data stored elsewhere, and code for linked lists, trees, or graphs is unreadable without knowing where a running Java program keeps its variables and objects.

1.4.1 The call stack and the heap

A running Java program uses two main memory regions. The first is the **call stack**. Each method call opens a **stack frame** there for its parameters and local variables; when the method returns, the frame and all its locals vanish. Frames pile up, and the last opened is the first closed — hence the name.

The second is **heap memory**. Every object created with `new`, every array, and every `String` lives there. Heap objects belong to no frame and can outlive the method that created them.

Definition 1.5

A **primitive variable** holds its value directly. A **reference variable** holds not the value of a heap object but its location, its address. That address is called a **reference**. A reference to no object has the value `null`.

Every Java type except the eight primitives is a reference type: arrays, `String`, our own classes, and every library class. Four lines show both kinds:

```
int x = 5;              // primitive: a value
int[] a = {7, 2, 9};    // reference → array
String s = "data";      // reference → object
int[] p = null;         // points to nothing
```

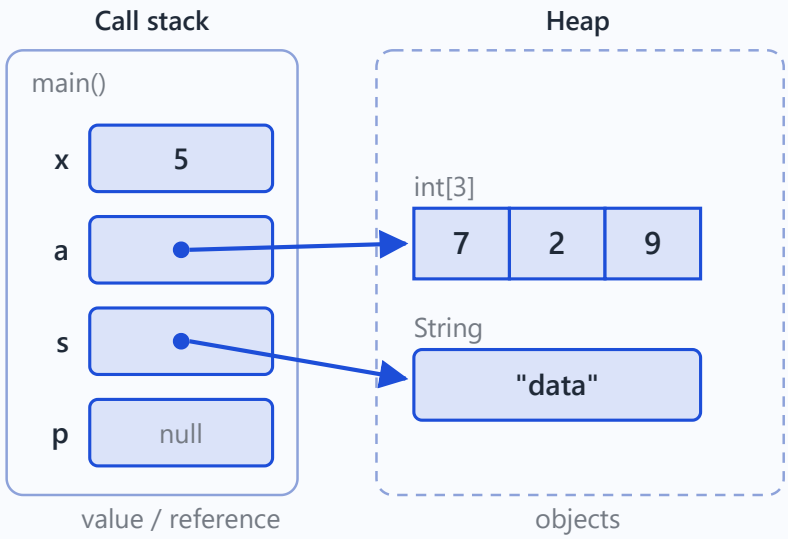


Figure 1.2: Memory after the four lines. In the frame of `main()`, `x` holds 5 directly. `a` and `s` are only arrows, references: one to a three-element `int` array on the heap, the other to a `String` object containing `"data"`. `p` is `null` and points to nothing.

In Figure 1.2, the four boxes in the frame are all small and equal in size: whether the array has 3 elements or 3 million, `a` holds only an address. The data is on the heap.

An object no reference points to is unreachable and wastes space. Java's **garbage collector** periodically reclaims such objects, so memory is never freed by hand; in C that falls to the programmer, and forgetting it leaks memory.

Using the object behind a `null` reference fails at run time. `p` points to no array, so `p[0]` finds no element and Java throws a `NullPointerException`, the most common error in work with linked structures.

Common mistake

The stack region of memory is not the stack data structure. The call stack is the stack Java uses to manage method calls, named for its last-in, first-out rule; the `Stack` interface later in these notes is a data structure for the programmer's own data. “Heap” has two unrelated meanings as well: the memory region and the data structure used for priority queues. The glossary calls the first *heap memory* and the second *heap*.

1.4.2 Code trace: value or reference?

Assignment (`=`) copies the value on the right into the variable on the left. What that value is depends on the variable, as this program shows:

```
int x = 5;
int y = x;
y = 7;
int[] a = {1, 2, 3};
int[] b = a;
b[0] = 9;
b = new int[2];
System.out.println(x + " " + a[0]);
```

Table 1.4 gives the variables and the heap after each line, with objects numbered #1, #2 in order of creation.

Line	`x`	`y`	`a`	`b`	Heap
1 `int x = 5;`	5	—	—	—	empty
2 `int y = x;`	5	5	—	—	empty
3 `y = 7;`	5	7	—	—	empty
4 `int[] a = {1, 2, 3};`	5	7	→#1	—	#1 = [1, 2, 3]
5 `int[] b = a;`	5	7	→#1	→#1	#1 = [1, 2, 3]
6 `b[0] = 9;`	5	7	→#1	→#1	#1 = [9, 2, 3]
7 `b = new int[2];`	5	7	→#1	→#2	#1 = [9, 2, 3], #2 = [0, 0]
8 `println`	5	7	→#1	→#2	output: `5 9`

Table 1.4: Trace of the value-versus-reference program. “→#1” means the variable points to object 1.

Lines 1–3. `x` is primitive; 5 sits directly in the frame. `int y = x;` copies it, giving two independent boxes holding 5, so `y = 7;` leaves `x` at 5.

Line 4. `{1, 2, 3}` creates a three-element array on the heap. `a` holds a reference to it, drawn as an arrow.

Line 5. `int[] b = a;` also copies, but what it copies is the value in `a` — an address. The array itself is not copied: the heap still holds one array, and `a` and `b` are two arrows to it.

Line 6. `b[0] = 9;` follows `b` to the object and changes cell 0. `a` points to the same object, so `a[0]` is now 9 too. Reaching one object through several references is called **aliasing**: `a` and `b` are two names for one array.

Line 7. `b = new int[2];` creates a new two-element array, which Java fills with 0s, and turns `b`'s arrow toward it. `a` is unaffected: the address in `b` changed, not object #1.

Line 8. `x + " " + a[0]` joins 5, a space, and 9: the output is `5 9`.

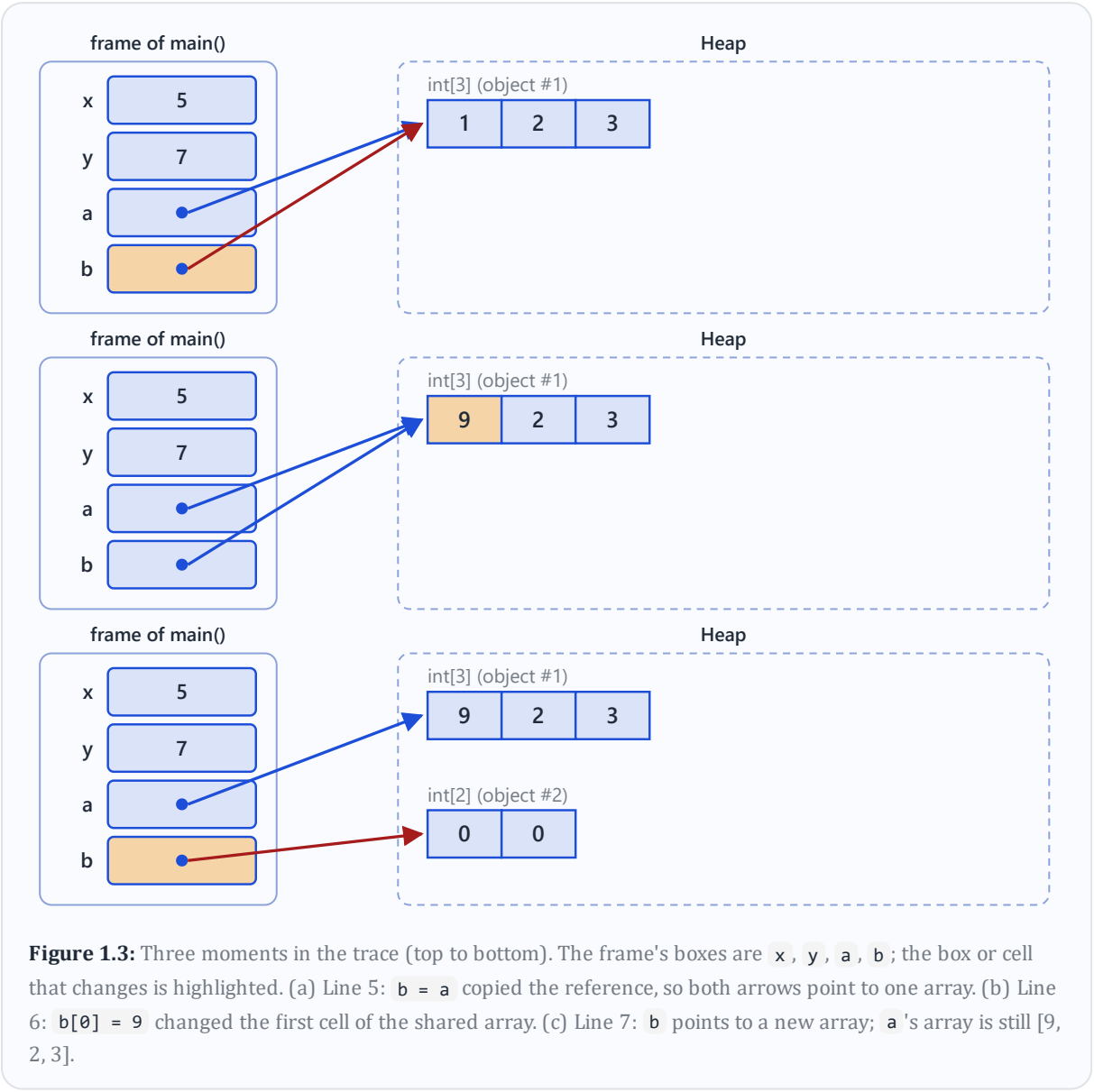


Figure 1.3: Three moments in the trace (top to bottom). The frame's boxes are `x`, `y`, `a`, `b`; the box or cell that changes is highlighted. (a) Line 5: `b = a` copied the reference, so both arrows point to one array. (b) Line 6: `b[0] = 9` changed the first cell of the shared array. (c) Line 7: `b` points to a new array; `a`'s array is still [9, 2, 3].

The whole trace reduces to one rule: `=` copies the *value* of a primitive and the *address* of a reference. Assigning to a reference variable redirects its arrow; assigning through it, as in `b[0] = …` or `obj.field = …`, changes the object. Our own classes behave the same way:

```
Student st1 = new Student("Alice", 101);
Student st2 = st1;         // a second reference, no new object
st2.id = 202;
System.out.println(st1.id);  // 202
```

`st2 = st1` creates no student, only a second arrow, and `st2.id = 202` changes the shared object, so `st1.id` is 202 too.

Common mistake

`int[] b = a;` does not copy the array. A real copy needs a new array filled element by element, which `int[] b = a.clone();` or `Arrays.copyOf(a, a.length)` does in one step. Even that copy is one level deep: if the elements are objects, only their references are copied. This is called a *shallow copy*.

1.4.3 Passing arguments to methods

In a method call, each argument is assigned to its parameter like an ordinary `=`. Java always **passes by value**: the parameter gets a copy — of the number for a primitive, of the address for a reference. Two swap methods show the consequence:

```
static void swap(int a, int b) {
  int tmp = a; a = b; b = tmp;
}
static void swap(int[] arr) {
  int tmp = arr[0]; arr[0] = arr[1]; arr[1] = tmp;
}
```

```
int p = 1, q = 2;
swap(p, q);
System.out.println(p + " " + q);              // 1 2
int[] nums = {1, 2};
swap(nums);
System.out.println(Arrays.toString(nums));    // [2, 1]
```

Example 1.6

What are `p`, `q`, and `nums` after the two calls?

1. **The call `swap(p, q)`:** A new frame gets copies: `a` = 1, `b` = 2. The method swaps them to `a` = 2, `b` = 1, and the frame is discarded on return. `p` and `q` were never touched: still 1 and 2.

2. **The call** `swap(nums)`**:** `arr` gets a copy of the address in `nums`, so two arrows point to one array, and swapping `arr[0]` and `arr[1]` swaps the shared cells.
3. **Result:** The output is `1 2` and `[2, 1]`. A method cannot change the caller's primitive variables, but through a reference it can change an object the caller points to.

The rule has a limit. Had the method run `arr = new int[2];`, only the local arrow would move; `nums` would keep the old array, and the unreferenced new one would be garbage-collected after the return. A method can change an object's contents but cannot point the caller's variable elsewhere.

Common mistake

The saying "Java passes primitives by value and objects by reference" predicts most outcomes but misstates the mechanism. Java passes everything by value; for a reference, the value is an address. The difference shows when a method assigns a new object to its parameter: under true pass-by-reference the caller's variable would change, and in Java it does not.

1.4.4 Strings and comparison

`String` is a reference type with one big difference from arrays: a `String`'s contents never change after creation. This is **immutability**. `toUpperCase()`, `substring()`, and `+` leave the object alone and return a new one.

```
String s1 = "course";
String s2 = s1;
s2 = s2.toUpperCase();
System.out.println(s1 + " " + s2);      // course COURSE
String u1 = new String("data");
String u2 = new String("data");
System.out.println(u1 == u2);           // false
System.out.println(u1.equals(u2));      // true
```

After two lines, `s1` and `s2` point to one `"course"` object. `s2.toUpperCase()` creates a new `"COURSE"` object, and the assignment turns `s2`'s arrow toward it. The old object is unchanged, so `s1` still prints `course`. An array's shared object could be changed (`b[0] = 9`); `String` has no such operation, so variables sharing a `String` cannot affect each other.

The last lines make a second point. Each `new String("data")` creates a separate heap object, so `u1 == u2`, which asks whether both arrows point to one object, gives `false`; `u1.equals(u2)` compares contents and gives `true`.

Common mistake

Strings must not be compared with `==`, which compares addresses for reference types. Java often stores identical literals from the source code as one object, so `==` sometimes seems to work, but for a string read from the keyboard or computed at run time it fails. Compare contents with `equals`.

1.5 Generics

A stack, list, or queue works the same whatever it holds; `push` and `pop` do not care whether the elements are integers or students. A stack class per element type wastes effort and copies every bug. Generics avoid this without giving up type safety.

1.5.1 The trouble with a non-generic box

Take the simplest "data structure," a box holding one item. To accept anything, it can use `Object`, the common ancestor of all Java classes:

```
public class ObjectBox {
  private Object item;
  public void put(Object x) { item = x; }
  public Object get() { return item; }
}
```

```
ObjectBox b = new ObjectBox();
b.put(42);                    // compiles: 42 is an Object too
String z = (String) b.get();  // compiles, but throws ClassCastException at run time
```

Accepting everything is exactly the problem. `put(42)` compiles. `get()` returns `Object`, so storing the result in a `String` needs a cast, `(String)`, which the compiler cannot verify and therefore accepts. Only at run time does the 42 turn out not to be a `String`: `ClassCastException`. By the standard of 1.3.3 this is the worst case, an error caught at run time rather than compile time.

1.5.2 Box<T>: one class for every type

Definition 1.6

A **generic** class takes one or more **type parameters**: names standing for types unknown when the class is written, usually a single capital letter such as `T`, `E`, `K`, or `V`. Code using the class supplies a concrete reference type for each. The language feature is called **generics**.

The generic box:

```
public class Box<T> {
  private T item;
  public void put(T x) { item = x; }
  public T get() { return item; }
}

Box<String> b = new Box<>();
b.put("data");
String s = b.get();     // no cast needed
Box<Integer> m = new Box<>();
m.put(42);              // int → Integer
// m.put("x");          // COMPILE-TIME ERROR
```

`<T>` after the class name declares the type parameter, and inside the class `T` acts as a real type: `item` is a `T`, `put` takes one, `get` returns one.

For `Box<String> b`, the compiler substitutes `String` for `T`: `b.put` accepts only a `String` and `b.get()` returns one, so `String s = b.get();` needs no cast. The empty brackets in `new Box<>()` are the *diamond* operator; the compiler infers the type argument from the left-hand declaration.

`Box<Integer> m` comes from the same class, for integers. Uncomment `m.put("x")` and compilation fails, since `put` on a `Box<Integer>` accepts only `Integer`. The error of 1.5.1 is now caught at compile time.

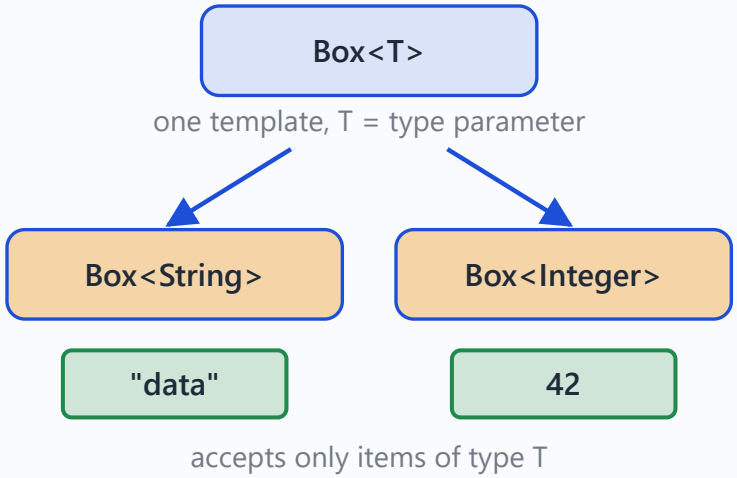


Figure 1.4: One `Box<T>` template yields the types `Box<String>` and `Box<Integer>`, each storing only its own type: `"data"` in one, 42 in the other.

As Figure 1.4 sums up, code written once serves as a separate type-safe class per type argument, with compile-time type checks, no casts, and no duplication.

Common mistake

A generic type should not be used without a type argument, as in `Box b = new Box();`. Java allows it for compatibility with old code and only warns, but the box then acts like the old `Object` box and type safety is gone. Always supply a type argument in angle brackets.

1.5.3 Wrapper classes and autoboxing

A type parameter must be a reference type; `Box<int>` is illegal. Inside a generic class, `T` values are always handled as references, and a primitive is not one — it holds its value directly.

The fix is Java's **wrapper classes**, one per primitive type. A wrapper is a small heap object carrying one primitive value: `Integer` for `int`, `Double` for `double`, `Character` for `char`, `Boolean` for `boolean`, and so on. A box of integers is therefore `Box<Integer>`.

Java wraps automatically. An `int` given where an `Integer` is expected is placed in an `Integer` object — **autoboxing**, which the comment "int → Integer" on `m.put(42)` refers to. Conversely, an `Integer` used in arithmetic or assigned to an `int` has its value extracted — **unboxing**.

Example 1.7

After 42 has been put into `Box<Integer> m`, the line `int sum = m.get() + 8;` runs. Which conversions happen?

1. `m.put(42)`: 42 is an `int`, but `put` expects an `Integer`. Java creates an `Integer` object holding 42 on the heap and stores its reference in `item` (autoboxing).
2. `m.get()`: returns a reference to that `Integer`.
3. `+ 8`: Addition works on primitives, so Java extracts 42 (unboxing) and computes 42 + 8 = 50.
4. **Assignment:** 50 is an `int` and goes straight into `sum`.

Autoboxing shortens code but sets two traps. The first is `null`: an empty `Box<Integer>` returns `null` from `get()`, and assigning that to an `int` makes Java try to unbox it and throw a `NullPointerException`. The second is comparison:

```
Integer i1 = 127, i2 = 127;
Integer i3 = 128, i4 = 128;
System.out.println(i1 == i2);         // true
System.out.println(i3 == i4);         // false
System.out.println(i3.equals(i4));    // true
```

`Integer` is a reference type, so `==` compares addresses. Java keeps ready-made objects for −128 to 127 and reuses them when boxing: both 127s point to one object and `==` gives `true`, while each 128 gets its own object and `==` gives `false`. `equals` compares contents and is right both times.

Common mistake

Comparing two `Integer`s from a generic structure with `==` is a well-hidden bug. Tests with small numbers pass; real data with a value above 127 then silently fails. Compare wrappers with `equals` or convert them to `int` first.

1.5.4 Java collections

Every data structure in Java's standard library is generic; together they are the **collections**. Two of the most used are `ArrayList<E>`, an ordered list of `E`s that grows as needed, and `Map<K,V>`, which maps `K` keys to `V` values and so takes two type parameters. This code uses three collections:

```
ArrayList<String> list = new ArrayList<>();
list.add("Ann"); list.add("Ben"); list.add("Ann");
HashSet<String> set = new HashSet<>(list);
HashMap<Integer, String> students = new HashMap<>();
students.put(101, "Alice");
students.put(102, "Bob");
students.put(101, "Carol");
System.out.println(list.size() + " " + set.size());     // 3 2
System.out.println(students.size() + " " + students.get(101));  // 2 Carol
System.out.println(students.get(103));                  // null
```

A list may hold duplicates, so three `add`s give size 3. A `HashSet` holds each element once; built from the list, it drops the second "Ann" and has size 2. `HashMap` keys are unique: a second `put` with the same key overwrites the old value instead of adding a pair, so 101 now maps to "Carol" and the map has 2 pairs. For a missing key (103), `get` returns `null`.

Type parameters protect this code too: given `students.put("101", "Alice")`, the compiler would see a `String` key where an `Integer` belongs and refuse. The structures we write in this course are generic classes in the same way.

1.6 Non-primitive data structures

A primitive holds one number, character, or logical value, but real programs handle hundreds of students, thousands of cities, millions of words, and need structures that keep many values together in a definite arrangement.

1.6.1 Composite structures

Structures built from primitive types and other structures are **non-primitive** data structures. By the relationship among their elements they fall roughly into these groups:

- **Linear:** arrays and lists. The elements form a sequence; each has one before it and one after it.
- **Linked:** linked lists. References connecting the elements build the sequence.
- **Hierarchical:** trees. Each element can have one above it and several below.
- **Network:** graphs. Each element can be related to any number of others.
- **Indexed:** hash tables. An element's position is computed from its key.
- Beyond these, sets, maps, and many more.

They share three traits. They are composite, holding many values, even of different types. Most are dynamic, growing and shrinking during a run; arrays are the exception, since a Java array's length is fixed at creation. And each is specialized: one returns the latest element fast, another finds a key fast, another suits finding paths.

1.6.2 Four values, eight structures

The clearest way to compare structures is to put the same data in each. Figure 1.5 arranges 7, 2, 9, 5 in eight ways, each with its own strengths and weaknesses.

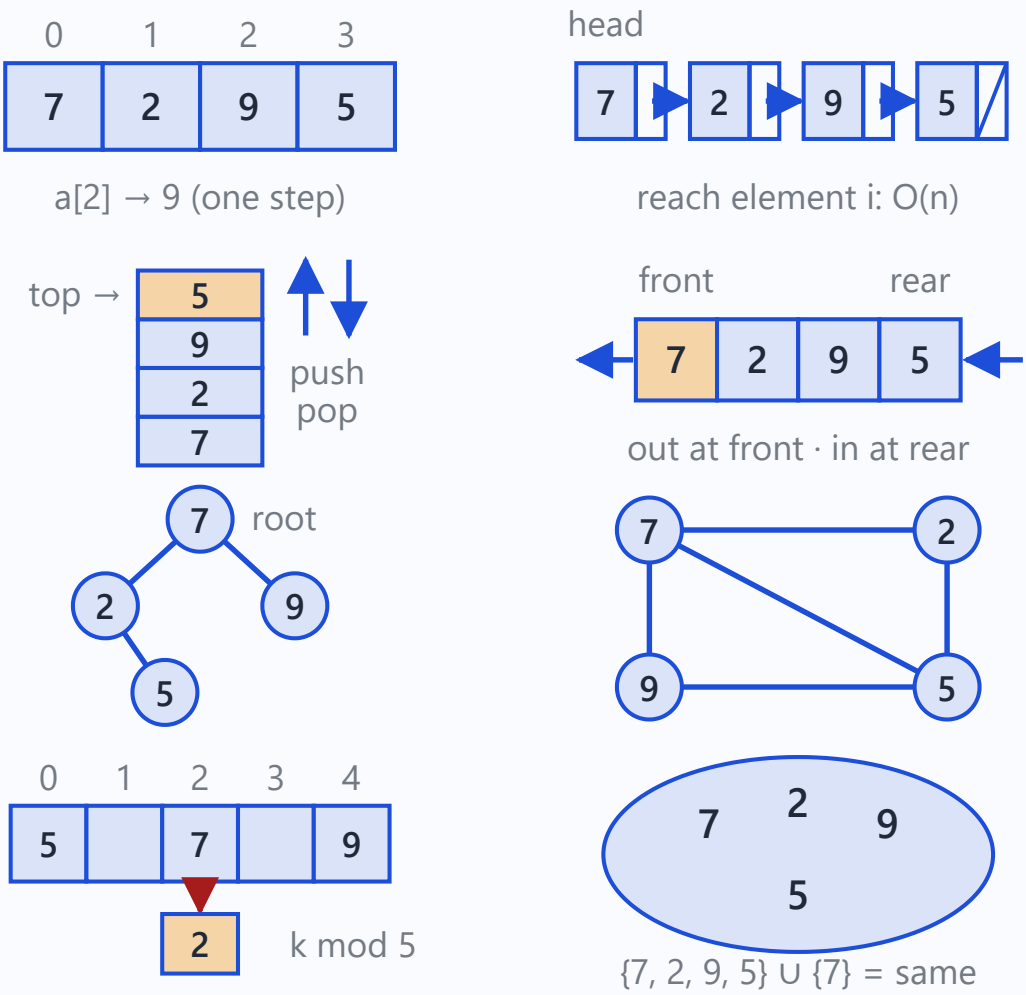


Figure 1.5: The same four values (7, 2, 9, 5) in eight arrangements, left to right and top to bottom: (a) array, (b) linked list, (c) stack, (d) queue, (e) tree, (f) graph, (g) hash table, (h) set.

(a) Array. The values occupy consecutive cells, indices 0 to 3. `a[2]` yields 9 in one step: by the address calculation of 1.2.3, indexed access is *O(1)*. But the size is fixed; a fifth value has no room.

(b) Linked list. Each value has its own node pointing to the next: 7 → 2 → 9 → 5 → `null`. One reference to the head suffices. Adding a node redirects a few arrows, and the size is flexible. The price: reaching element *i* means following *i* arrows from the head, *O(n)*.

(c) Stack. The values are piled with 7 at the bottom and 5, the latest, on top. Insertion (`push`) and removal (`pop`) happen only at the top: **LIFO** (last in, first out). The next `pop` returns 5.

(d) Queue. The values line up by arrival, 7 in front and 5 at the rear. Elements join at the rear and leave from the front: **FIFO** (first in, first out). The next removal returns 7. Stack and queue release the same values in opposite orders.

(e) Tree. The values form a hierarchy: 7 is the root, 2 and 9 its children, 5 a child of 2. The childless 9 and 5 are leaves. Exactly one path leads from the root to each node.

(f) Graph. The values are nodes and the lines edges; each node connects to two or three others. Unlike a tree, a graph need not be hierarchical: a node can be reached by several paths, and edges can form cycles. The relationship is many-to-many.

(g) Hash table. The values go into a 5-cell array at positions given by a **hash function** $h(k)$, here $h(k) = k \bmod 5$, the remainder on division by 5: 5 mod 5 = 0, 7 mod 5 = 2, 9 mod 5 = 4. But 2 mod 5 is also 2, and cell 2 already holds 7. Two values in one cell is a **collision**, and hash tables have extra rules to resolve it. With few collisions, a search just computes the cell and looks there, so it takes $O(1)$ on average.

(h) Set. A set holds distinct elements in no particular order. Adding 7 again changes nothing: {7, 2, 9, 5} ∪ {7} is the same set. Its close relative, the **map**, attaches a value to each distinct key (key → value).

Example 1.8

With the same 5-cell table and $h(k) = k \bmod 5$, we add 13 and 20 after 7, 2, 9, 5. Where do they go?

1. **13:** 13 = 5 · 2 + 3, remainder 3. Cell 3 is empty; 13 goes there.
2. **20:** 20 = 5 · 4 + 0, remainder 0. Cell 0 holds 5: a collision.
3. **Observation:** 6 values now sit in 5 cells with 2 collisions (2 with 7, 20 with 5). Collisions multiply as a table fills, so hash tables move to a larger table once a certain fill ratio is passed.

Common mistake

No structure is best at everything. An array is unbeatable at indexed access but slow at inserting in the middle; a linked list is the reverse. A hash table finds keys very fast but cannot list elements in sorted order. Each of the eight gives up some operations to make others cheap.

1.7 Abstract data types and data structures

“Stack” can mean a behavior — add and remove at the top, last in, first out — or that behavior built in memory from a particular array, index variable, and code. Keeping the two levels apart is the key to understanding and designing data structures.

1.7.1 What versus how

Definition 1.7

An **abstract data type** (ADT) specifies which operations a kind of data supports and how they behave, not how they are carried out. A **data structure** is a concrete structure implementing an ADT: it fixes how the data sits in memory and the steps of each operation.

Lists, stacks, queues, sets, maps, and graphs are ADTs, descriptions of behavior. Linked lists, arrays, and adjacency matrices are data structures, ways of building ADTs in memory and concerned with performance.

Think of a car. It offers a steering wheel and two pedals, and anyone who knows that "interface" can drive any car; whether the engine is gasoline or electric, or how the brakes work, does not concern the driver. Wheel and pedals are the ADT, engine and brakes the data structure. Different mechanisms can provide the same interface, and the driver can switch cars without noticing.

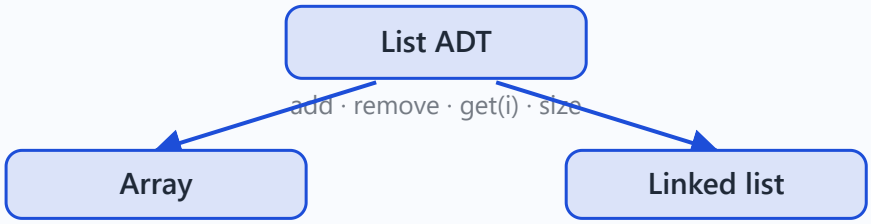


Figure 1.6: The List ADT (operations add, remove, get(i), size) can be implemented with two different data structures: an array and a linked list.

Figure 1.6 shows this for lists. The List ADT offers four operations — add, remove, get the *i*th element, report the size — and both an array and a linked list can provide them. They differ only in cost: an array does `get(i)` instantly but shifts everything to add at the front; a linked list adds at the front instantly but walks from the start for `get(i)`. Table 1.5 compares the two concepts.

	Abstract data type (ADT)	**Data structure**
Question it answers	what is done?	how is it done?
Focus	operations and behavior	memory layout and performance
Examples	list, stack, queue, set, map, graph	array, linked list, adjacency matrix
In Java	interface (`interface`)	class (`class`)
How many	one ADT	several implementations of the same ADT

Table 1.5: Abstract data types and data structures compared.

Common mistake

"A stack is a data structure, and so is an array" wrongly puts the two on one level. A stack is an ADT, pure behavior; an array is a memory layout, one way to build a stack. Everyday speech calls both "data structures," but a design asks two questions: which ADT is needed, and which data structure implements it.

1.7.2 From ADT to Java: the interface

Java has a language feature for each side of the distinction. An **interface** records an ADT's contract: which methods exist and what each takes and returns, with no bodies. The stack ADT's interface:

```
public interface Stack<E> {
  int size();         // number of elements
  boolean isEmpty();  // is the stack empty?
  void push(E e);     // add on top
  E top();            // return the top element
  E pop();            // remove the top element
}
```

`size()` returns the element count, `isEmpty()` whether there are none, and `push(e)` puts `e` on top. `top()` returns the top element without removing it; `pop()` removes and returns it. The interface is generic: `E` is the element type, so one interface serves all types.

With no fields and no bodies, the interface does not say whether elements live in an array or in linked nodes. A **class** says that: one declaring `implements Stack<E>` must give all five methods bodies. The `interface` is the contract, what to do; the `class` is the data structure, how. An ADT can have several **implementations**.

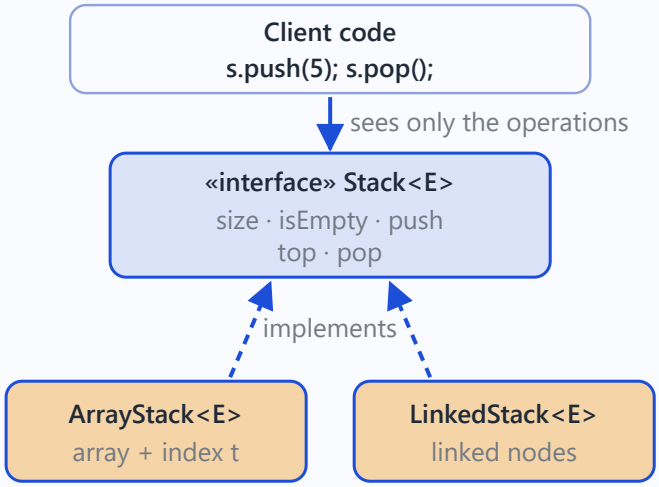


Figure 1.7: Client code uses only the `Stack<E>` interface and sees only the operations. Two classes implement it: `ArrayStack<E>`, with an array and an index `t`, and `LinkedStack<E>`, with linked nodes.

Code that uses the stack is **client code**, and Figure 1.7 shows that it depends only on the interface. The client creates a stack like this:

```
Stack<Integer> s = new ArrayStack<>(10);
```

The variable's type is the interface (`Stack<Integer>`); the object comes from a specific class (`ArrayStack`). Later code can use only interface operations such as `s.push(…)` and `s.pop()`, so switching to a linked-node stack changes only this line, `new ArrayStack<>(10)` becoming `new LinkedStack<>()`.

Common mistake

Declaring the variable with the class type, as in `ArrayStack<Integer> s = …`, works but ties the client to the class. If the client used a class-specific method (a debugging method, say), changing the implementation affects all client code. An interface-typed variable ties the client only to the contract.

1.7.3 An implementation: ArrayStack

An array-based stack fills the array from the start: the first push goes into cell 0, the second into cell 1. The top is the last filled cell, so one variable holding its index suffices; we call it `t` (for top).

```
public class ArrayStack<E> implements Stack<E> {
  private E[] data;         // the elements
  private int t = -1;       // index of the top element
  public ArrayStack(int capacity) {
    data = (E[]) new Object[capacity];
  }
  public int size() { return t + 1; }
  public boolean isEmpty() { return t == -1; }
  public E top() {
    if (isEmpty()) return null;
    return data[t];
  }
  // push and pop
}
```

Piece by piece:

- `implements Stack<E>`: the class accepts the stack contract; its type parameter `E` becomes the interface's `E`.
- `private E[] data`: the elements. `private` blocks outside access, forcing the client through the interface and so protecting top-only access.
- `private int t = -1`: the top element's index. An empty stack has no top, signaled by −1, which is not a valid index.
- The constructor creates an array of the requested capacity. `new E[capacity]` is illegal, because the type parameter is unknown at run time and Java cannot create an array of an unknown type; instead an `Object` array is cast with `(E[])` and assigned to `data`. The compiler warns of an "unchecked cast," but it is safe: the array never leaves the class and receives only `E`s.
- `size()`: the elements fill cells 0 through `t`, so there are `t + 1`; for an empty stack, −1 + 1 = 0.
- `isEmpty()`: the stack is empty when `t == -1`.
- `top()`: returns `null` for an empty stack, otherwise the top element, `data[t]`, without removing it; `t` is unchanged.

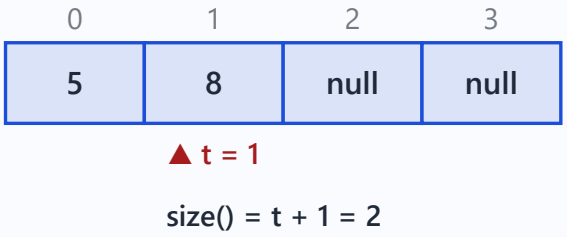


Figure 1.8: An `ArrayStack` of capacity 4: the cells hold 5, 8, `null`, `null`, and `t = 1`. The top element is 8, and `size() = t + 1 = 2`.

In Figure 1.8, 5 and then 8 have been pushed: `t` = 1 marks the 8, and cells 2 and 3 wait empty.

> **Common mistake**
>
> `t` is the top element's index, not the element count, which is `t + 1`. Confusing them causes off-by-one errors in `size()` and in the full check. Some books keep a count instead, marking an empty stack with 0; both conventions work, but they must not be mixed in one class.

1.7.4 push and pop

The two missing methods change the stack. The slides show them with a short client:

```
public void push(E e) {
  if (size() == data.length)
    throw new IllegalStateException("full");
  data[++t] = e;
}
public E pop() {
  if (isEmpty()) return null;
  E e = data[t];
  data[t--] = null;     // clear for GC
  return e;
}
s.push(5); s.push(8);
System.out.println(s.pop());
s.push(3);
```

`push(e)`. If the element count equals the array length, there is no room and the method throws an `IllegalStateException`; pushing onto a full fixed-capacity stack is **overflow**. Otherwise `data[++t] = e` increments `t`, then writes at the new index. `data[t++] = e` would overwrite the old top.

`pop()`. On an empty stack the method returns `null`; otherwise it saves the top element in `e`. `data[t--] = null` writes `null` into the cell at the old `t`, then decrements `t`. Clearing the cell helps the garbage collector: a cell still pointing to the old element would keep it alive after the client was done with it. Finally the saved element is returned.

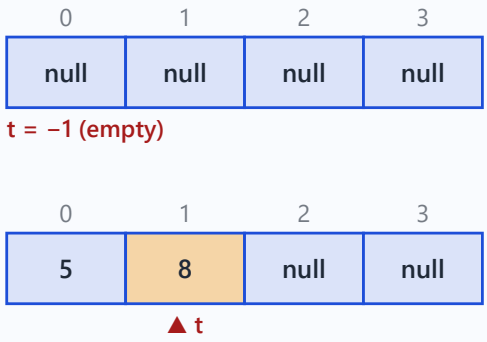
> **Common mistake**
>
> Clearing before reading in `pop`, as in `data[t--] = null; return data[t];`, returns the wrong element. The first line deletes the element and decrements `t`, so the second returns the one below. The right order is to save the element, then clear the cell and decrement `t`, then return what was saved.

1.7.5 Code trace: ArrayStack

The client runs on an empty `ArrayStack` of capacity 4: `push(5)`, `push(8)`, print `pop()`, `push(3)`. Table 1.6 gives the call stack, `t`, and the array per step. Line numbers refer to the slides' code block: `push` is lines 1–5, `pop` 6–11, the client 12–14.

Step	Line	Call stack	`t`	`data`	Explanation
1	12	main	−1	[null, null, null, null]	`push(5)` is called
2	2	main | push(5)	−1	[null, null, null, null]	`size()` = 0 ≠ 4: not full
3	4	main | push(5)	0	[5, null, null, null]	`++t`, then `data[0] = 5`
4	12	main	0	[5, null, null, null]	`push(8)` is called
5	2	main | push(8)	0	[5, null, null, null]	`size()` = 1 ≠ 4
6	4	main | push(8)	1	[5, 8, null, null]	`data[1] = 8`
7	13	main	1	[5, 8, null, null]	`pop()` is called
8	7	main | pop()	1	[5, 8, null, null]	`t` = 1: not empty
9	8	main | pop()	1	[5, 8, null, null]	`e = data[1]` = 8
10	9	main | pop()	0	[5, null, null, null]	cell cleared, `t--`
11	10	main | pop()	0	[5, null, null, null]	`return 8`
12	13	main	0	[5, null, null, null]	`8` is printed
13	14	main	0	[5, null, null, null]	`push(3)` is called
14	2	main | push(3)	0	[5, null, null, null]	`size()` = 1 ≠ 4
15	4	main | push(3)	1	[5, 3, null, null]	`data[1] = 3`
16	14	main	1	[5, 3, null, null]	final state

Table 1.6: `ArrayStack` code trace (capacity 4).



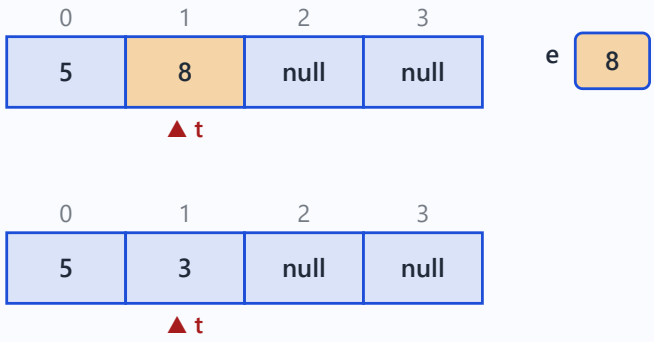


Figure 1.9: Four moments in the trace (left to right, top to bottom). (a) Start: four `null` cells, `t = -1`. (b) After two `push`es: [5, 8], `t = 1`. (c) Inside `pop`, `e = data[1]` = 8 has been saved. (d) Final state after `push(3)`: [5, 3], `t = 1`.

Three things stand out. Each call opens a frame ("main | push(5)") that closes on return — the memory model of 1.4.1 at work. `pop` returns 8, the latest element, while 5, the first, stays put (LIFO). And `push(3)` reuses the just-cleared cell 1: nothing ever shifts; only `t` moves.

Example 1.9

An empty `ArrayStack<String>` of capacity 2 receives `push("a")`, `push("b")`, `push("c")`. What happens? And what do `pop()` and `top()` return on an empty stack of capacity 3?

1. `push("a")`: `size()` = 0 ≠ 2; `t = 0`, `data = ["a", null]`.
2. `push("b")`: `size()` = 1 ≠ 2; `t = 1`, `data = ["a", "b"]`. The array is full.
3. `push("c")`: `size()` = 2 = `data.length`, so the method throws an `IllegalStateException` with the message `"full"`; the stack is unchanged.
4. **Empty stack:** `t = -1`, so `isEmpty()` is `true`, and `pop()` and `top()` both return `null`.

The last step reflects a design choice: this class reports popping an empty stack (**underflow**) by returning `null` rather than as an error, where some libraries throw an exception. Both are common, but the client of a `null`-returning stack must check results for `null`. And if `null` was itself pushed, a `null` from `pop` cannot tell "empty" from "the top was `null`."

1.7.6 What the operations cost

Operations are compared by the work they do, usually in **Big O** notation. Two meanings suffice for now. *O(1)*: the number of steps is constant, independent of the element count. *O(n)*: the steps grow in proportion to the element count *n*, so doubling the elements roughly doubles the work.

Operation	Time
`size`, `isEmpty`, `top`	O(1)

Operation	Time
`push`, `pop`	O(1)
Memory	O(N) (capacity)

Table 1.7: Cost of the `ArrayStack` operations (*N*: capacity).

Every operation in Table 1.7 is *O(1)*, as the code shows. `size` does one addition, `isEmpty` one comparison, `top` a comparison and an array access; `push` a comparison, an increment, and a write; `pop` a comparison, a read, a write, and a decrement. No method loops, and array access is one step (1.2.3). With 3 elements or 3 million, the step count is the same.

Memory, though, is proportional to capacity: a stack of capacity *N* reserves *N* cells however few it holds. That is the array's price. The capacity is fixed and `push` throws once it is reached; too large wastes memory, too small fills early. The remedies are to move to a larger array when full or to replace the array with linked nodes. Either way the stack ADT stays the same and only the class changes — the practical payoff of 1.7.1.

1.8 Organizing data and choosing a structure

Before choosing a structure, look at how the data items relate. Are they independent, or do they form a sequence, a hierarchy, or a tangled network? The answer narrows the candidates considerably.

1.8.1 Four ways to organize data

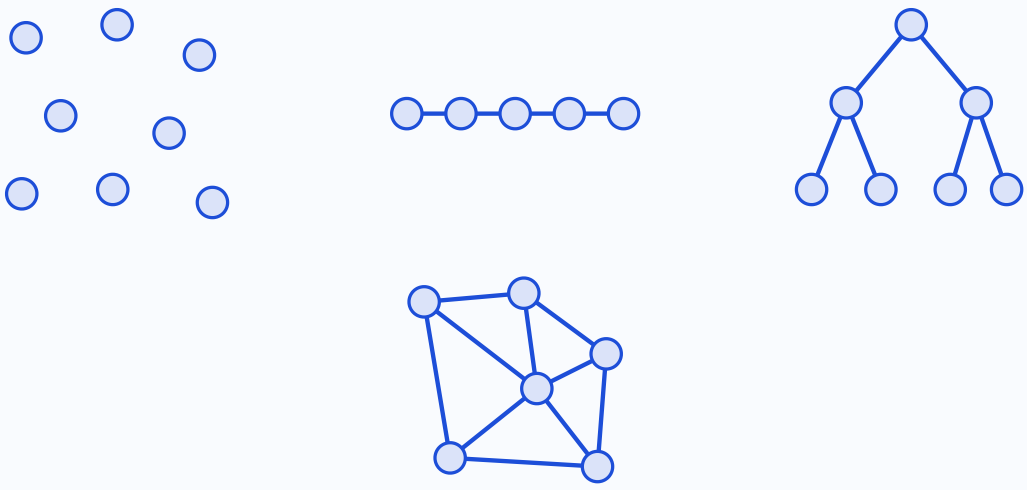


Figure 1.10: Four ways of organizing data (left to right): (a) unordered: scattered, unconnected elements; (b) linear: a straight chain; (c) tree: a hierarchy branching from a root; (d) graph: elements joined by multiple connections.

Figure 1.10 shows the four basic arrangements; Table 1.8 gives the relationship each sets up and an everyday example.

Organization	Relationship	Everyday example
Unordered	none	students in a classroom
Linear	one-to-one	students waiting in line
Tree	one-to-many	family tree
Graph	many-to-many	road map

Table 1.8: Ways of organizing data.

In an **unordered** arrangement the elements are unrelated: a classroom's students form a group, but “the student after Alice” means nothing. The questions concern membership — is Bob in this class, how many students are there? Sets and maps fit.

In a **linear** arrangement each element has at most one predecessor and one successor: one-to-one. In a cafeteria line everyone has one person ahead and one behind, except the first and last. Arrays, lists, linked lists, stacks, and queues are linear; they differ in which ends allow insertion and removal.

In a **tree** each element has at most one parent above it but may have several children: one-to-many. Family trees, organization charts, and computer folders follow this pattern; each folder has one parent folder but may hold many subfolders.

In a **graph** each element can relate to any number of others: many-to-many. On a road map, roads lead from each city to many others. Social-network friendships and computer networks are graphs too. The graph is the most general arrangement, with linear arrangements and trees as restricted cases.

Example 1.10

Which arrangement fits each of the following?

1. **A book's chapters, sections, and subsections:** Each subsection belongs to one section and each section to one chapter, while a chapter has many sections: one-to-many, a tree. These notes' table of contents is such a tree.
2. **The stops on a bus route:** Each stop has a previous and a next stop: linear.
3. **Prerequisites among university courses:** A course can have several prerequisites and be a prerequisite of several courses: many-to-many, a graph.
4. **A library's members:** No order or hierarchy; all that matters is whether someone is a member: unordered.

1.8.2 Choosing a data structure

The right structure follows from how the data will be used. These questions decide it:

- **Can the structure be filled completely at the start?** If all the data is known in advance (the twelve month names, say), it is built once and then only read; data arriving over time needs room to grow.
- **Can elements be deleted?** If not, the structure can be much simpler. Frequent deletion calls for a cheap way to close the gap a deleted element leaves.
- **Does insertion order matter?** Processing in arrival order suggests a queue; latest-first, a stack. If order is irrelevant, consider a set or a map.
- **Can the elements be sorted?** If they can be ordered by size or alphabetically, fast methods such as the halving search of 1.2.1 apply.
- **Are elements processed in a fixed order, or is random access needed?** Frequent access by position ("give me element 5") calls for an array; for front-to-back passes a linked list will do.

In short: **static** structures for unchanging data, **dynamic** ones for frequent insertion and deletion. A static structure, such as a fixed-size array, gives the fastest access; a dynamic one, such as a linked list, adapts to change but accesses more slowly.

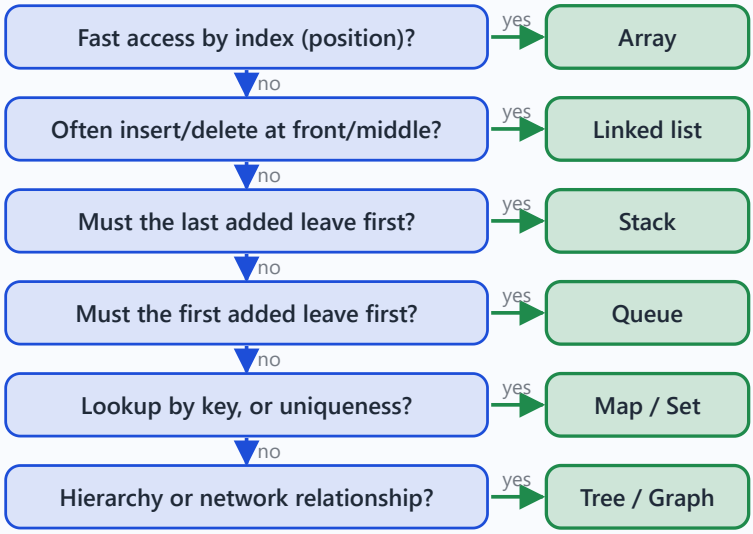


Figure 1.11: A question flow for choosing a data structure. The questions are asked top to bottom, and the first "yes" suggests a structure: fast access by index → array; frequent insertion or deletion at the front or middle → linked list; last added leaves first → stack; first added leaves first → queue; lookup by key or uniqueness → map or set; hierarchy or network → tree or graph.

Figure 1.11 turns the questions into a flow. It is a starting point, not a rule: real problems often answer "yes" to several questions and then use several structures together.

Example 1.11

A dictionary app must show a typed word's meaning at once and list the 20 most recently searched words, latest on top. Which structures fit?

1. **Word → meaning:** Words have no index, and middle insertion is not the main need, but "lookup by key?" is a yes: the word is the key, its meaning the value. A map fits; a hash table gives *O(1)* lookup on average.
2. **Recent searches:** The latest must be on top, so "must the last added leave first?" is a yes. The ADT is a stack, and the limit of 20 suggests a fixed capacity.
3. **A subtle point:** Past the limit, the *oldest* word must go, but a pure stack removes only from the top. A structure working at both ends fits better. The flow's first "yes" is a start; always recheck the full requirements.

Common mistake

Picking the most familiar structure, usually an array or `ArrayList`, for every problem can be costly. Asking a list "have we seen this word before?" means scanning it end to end each time; for a million-word text this can approach the 10^{12} comparisons of 1.1.3. A set answers in constant time on average.

1.9 The fundamental data structures at a glance

Table 1.9 gathers the fundamental structures met so far, with each one's core idea and typical uses.

Structure	Core idea	Typical use
Array	consecutive cells; O(1) access by index, fixed size	tables and matrices, the basis of other structures
Linked list	nodes linked by references; flexible size	frequent insertion and deletion
Stack	LIFO: add / remove only at the top	undo, the call stack, bracket matching
Queue	FIFO: add at the rear, remove at the front	print queues, job queues, BFS
Tree	hierarchy: root, children, leaves	file systems, HTML documents (DOM), indexes
Priority queue	the highest-priority element always in front (heap)	task scheduling, emergency-room triage
Map / hash table	key → value; h(k) picks the bucket, O(1) on average	student ID → record, caching, word counts
Set	distinct elements; fast membership test	removing duplicates, union and intersection
Graph	nodes + edges; many-to-many relationships	road maps, social networks, shortest paths

Table 1.9: The fundamental data structures at a glance.

Each use in the table follows from a basic property of the structure.

The **array** underlies the others: stacks, queues, hash tables, and priority queues are often arrays inside. Images, game boards, and tables are stored as two-dimensional arrays (matrices).

The **linked list** suits changing sizes and frequent insertion at the front or middle: a music player adds or removes playlist songs without moving the others.

The **stack** embodies "undo the latest action first": an editor pushes each action, and undo pops one and applies its inverse. The call stack of 1.4.1 follows the same rule. In bracket matching, opening brackets are pushed and each closing bracket must match the one on top.

The **queue** is fair waiting. Print jobs, server requests, and tasks waiting for the operating system are handled in arrival order. Breadth-first search (BFS) also takes its next node from a queue.

The **tree** is hierarchy: computer folders, a web page's HTML elements (the DOM, Document Object Model, in browsers), and database indexes for fast search.

The **priority queue** removes by priority, not arrival. An emergency room sees patients by severity, not by when they came. It is usually built on a special tree called a heap.

The **map** and **hash table** lead quickly from a key to a value: from a student ID to a record, from a word to its count in a text. A browser's cache maps addresses to page contents.

The **set** removes duplicates and answers "is this element present?" quickly, with direct support for union and intersection.

The **graph** models the most general relationships. A maps app stores cities or intersections as nodes and roads as edges and searches for the shortest path between two points. With numbers such as distances on the edges, it is a weighted graph.

Example 1.12

Choose structures for these requirements of a multiplayer online game.

1. **Player information by name:** lookup by key → map (hash table).
2. **Players waiting to join, in arrival order:** FIFO → queue.
3. **Map regions and the passages between them:** each region connects to many others → graph.
4. **Item types in an inventory, without repeats:** uniqueness → set.
5. **Guild structure (leader, officers, members):** hierarchy → tree.

One application, five structures. That is the rule, not the exception: real software picks a structure per requirement and combines them.

1.10 Common mistakes

Table 1.10 collects the chapter's mistakes. Most come from forgetting whether a variable holds a value or a reference, or from misplacing the type that decides a result.

Mistake	Result	Fix
Computing a ratio as `7 / 30 * 100`	integer division: 0	multiply first, or write `7.0`
A sum beyond the `int` range	silent overflow, negative result	`long` or `Math.addExact`
Comparing `double`s with `==`	`0.1 + 0.2 == 0.3` is false	check that the difference is tiny
Copying an array with `b = a`	two names, one array (aliasing)	`a.clone()` or `Arrays.copyOf`
Expecting a new object assigned to a parameter to reach the caller	the caller's variable is unchanged	return the result, or change the object's contents
Comparing `String`s or `Integer`s with `==`	addresses are compared	`equals`
Writing `Box<int>`	compile-time error	`Box<Integer>`
Assigning a `null` `Integer` to an `int`	`NullPointerException`	check for `null` first
Declaring the variable with the class type	client tied to the implementation	interface type: `Stack<E>`
Taking `t` for the element count	off-by-one errors	`size() = t + 1`
Writing `data[t++] = e`	the top element is overwritten	`data[++t] = e`
Clearing before reading in `pop`	the wrong element is returned	read first, then clear

Table 1.10: Common mistakes in this chapter and their fixes.

1.11 Summary

- **Digital data** is text, numbers, images, and sound as 0s and 1s; an ADC turns signals into numbers and a DAC turns them back. n bits give 2^n values. Text uses character encodings (ASCII, Unicode), floating-point numbers IEEE 754, images pixel RGB values, sound sampling.
- An **algorithm** is a plan of steps for a task; a **data structure** organizes and stores data: Algorithms + Data Structures = Programs (Wirth). Each structure makes some operations cheaper, others dearer. The building blocks are objects, arrays, and links.
- A **data type** is values plus operations. Java's eight **primitive types** have fixed sizes and wrap around past the limit; `int` divided by `int` is an `int`. **Type safety** catches type violations at compile time.

- Local variables live in **call stack** frames, objects in **heap memory**. A primitive variable holds a value, a **reference** an address; `=` copies whichever it holds, so two references can share an object (aliasing). Parameters pass by value. The garbage collector reclaims unreachable objects.
- **Generics** (`Box<T>`) give one class for every reference type, with type errors caught at compile time. Type parameters cannot be primitive: `Integer` replaces `int`, and autoboxing converts. Java's collections are generic.
- An **abstract data type** says what is done (behavior); a **data structure** says how (memory layout). In Java an ADT is an **interface**, a data structure a **class** implementing it. With interface-typed variables, changing the implementation is a one-line edit.
- `ArrayStack` uses an array and a top index `t`: `t = -1` when empty, `size() = t + 1`. `push` increments, then writes; `pop` reads, then clears. All operations are *O(1)*; memory grows with capacity, and a full stack makes `push` throw.
- Data is unordered, linear, a tree, or a graph. Access pattern, insertion and deletion frequency, need for order, and kind of relationship decide the structure: static for stable data, dynamic for changing data.

Topic	Key points
Digital data	text, numbers, images, and sound are 0s and 1s under different encodings
Data type	set of values + operations; type safety catches errors at compile time
Primitive / reference	a primitive variable holds a value; a reference points to a heap object, and assignment copies only the reference
Generics	`Box<T>`: one class for every reference type; `Integer` instead of `int`
ADT / data structure	the ADT says what is done (interface), the data structure how (class)
Choosing a structure	decided by access pattern, insertion / deletion frequency, need for order, and kind of relationship

Table 1.11: Chapter summary.

1.12 Key terms

Each term links to the course glossary, which gives its Turkish equivalent and its use in other chapters.

digital data · algorithm · data structure · data type · primitive type · non-primitive · reference · null · heap memory · call stack · stack frame · shallow copy · immutable · generic · generics · autoboxing · unboxing · collection · abstract data type (ADT) · interface · array · index · capacity · linked list · node · stack · top · overflow · underflow · queue · first in, first out (FIFO) · tree · priority queue · map · hash table · hash function · collision · set · graph · static array · Big O

1.13 Test Yourself

1. How many bits do 1,000 different values need? How is 200 written in eight bits, and what is `(byte) 200` in Java?

   Answer

   Ten: 9 bits give only $2^9 = 512$ values, 10 bits $2^{10} = 1{,}024$. Since 200 = 128 + 64 + 8, it is `11001000`. A `byte` is signed and holds at most 127, so the leading 1 makes it negative: 200 − 256 = −56.

2. For `int a = 9;`, what do `a / 4 * 4` and `a * 4 / 4` give, and why do they differ?

   Answer

   8 and 9, evaluating left to right. In the first, integer division makes `9 / 4` 2, losing the fraction, and 2 · 4 = 8. In the second, 9 · 4 = 36 comes first, and 36 / 4 = 9 exactly.

3. After these lines, which arrays do `r` and `r2` point to?

   ```
   int[] r = {3, 4, 5};
   int[] r2 = r;
   r2[2] = r2[0] + r2[1];
   r2 = new int[]{0, 0, 0};
   r2[0] = 1;
   ```

   Answer

   `r`: `[3, 4, 7]`, `r2`: `[1, 0, 0]`. Line 2 copies the reference; line 3 writes 3 + 4 = 7 into the shared array, changing `r` too. Line 4 points `r2` to a new array, and line 5 changes only that.

4. After `StringBuilder sb = new StringBuilder("ab"); StringBuilder sb2 = sb; sb2.append("c");`, what does `sb` contain? What would change with `String` instead of `StringBuilder` and `sb2 = sb2 + "c"` instead of `append`?

   Answer

   `"abc"`. A `StringBuilder` is mutable: `sb` and `sb2` share one object, and `append` changes it. A `String` is immutable: `sb2 + "c"` creates a new object and the assignment only redirects `sb2`, so `sb` would stay `"ab"`.

5. After `Box<Double> d = new Box<>(); d.put(2.5);`, what does `double x = d.get() * 2;` give?

Does `d.put(3);` compile?

Answer

5.0: 2.5 was autoboxed into a `Double` and unboxed for the multiplication. `d.put(3)` does not compile: the `int` 3 can be boxed only into an `Integer`, which is not a `Double`. Write `d.put(3.0)`.

6. An empty `ArrayStack<Integer>` of capacity 3 receives, in order, `push(4)`, `push(6)`, `pop()`, `push(1)`, `top()`, `pop()`, `push(2)`, `push(8)`. What do the value-returning calls return? What are `t` and `data` at the end? What does a further `push(5)` do?

Answer

6, 1, 1: the first `pop` returns 6, `top` returns 1 without removing it, the second `pop` returns 1. Pushing 2 and 8 leaves `t = 2`, `data = [4, 2, 8]`, full. `push(5)` finds `size()` = 3 = `data.length` and throws an `IllegalStateException`.

7. Of queue, array, map, linked list, set, and adjacency matrix, which are abstract data types and which are data structures?

Answer

Queue, map, and set are ADTs, describing possible operations; array, linked list, and adjacency matrix are data structures, fixing how data is held in memory. A queue ADT can be built on an array or a linked list, and a graph ADT stored in an adjacency matrix.

8. Which structure suits each? (a) callers waiting at a call center, (b) a company's organization chart, (c) the number of distinct visitors to a website, (d) prerequisite relationships among courses.

Answer

(a) Queue: callers are served in arrival order (FIFO). (b) Tree: one manager per employee, many employees per manager (one-to-many). (c) Set: repeat visitors count once; the set's size is the answer. (d) Graph: a course can have several prerequisites and several successors (many-to-many).

9. The values 10, 24, 3, and 17 go into a 7-cell table with *h(k) = k mod 7*. Which cells? And in a 5-cell table with *k mod 5*?

Answer

Mod 7, all four land in cell 3: 10 = 7 + 3, 24 = 21 + 3, 3 = 0 + 3, 17 = 14 + 3. Every insertion after the first collides, so lookups lose constant time. Mod 5, the cells are 0, 4, 3, 2: no collisions. How evenly the hash function spreads values decides the table's speed.

10. A client's 500-line program begins with `Stack<String> s = new ArrayStack<>(100);`. To escape the capacity limit, the client wants to switch to `LinkedStack`, which uses linked nodes. How many lines change? What if the variable had been declared as `ArrayStack<String>`?

    Answer

    Only the first: `new ArrayStack<>(100)` becomes `new LinkedStack<>()`; the rest uses only the five interface operations. With a class-typed variable the declaration would change too, and lines using `ArrayStack`-specific methods would need rewriting.

1.14 Further reading

- M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, *Data Structures and Algorithms in Java* (6th ed., 2014), Chapters 1 and 2: Java's basic types, references, classes and interfaces, and generics, with code close to the stack example in these notes.
- R. Sedgewick and K. Wayne, *Algorithms* (4th ed., 2011), Sections 1.2 and 1.3: data abstraction, the separation of API from implementation, and generic stack and queue classes, through short, clear examples.
- N. Wirth, *Algorithms + Data Structures = Programs* (1976): the classic source for the idea that algorithms and data structures must be designed together; its first chapter presents the fundamental data types and structures from a historical perspective.

1.15 References

1. M. T. Goodrich, R. Tamassia, M. H. Goldwasser, *Data Structures and Algorithms in Java*, 6th ed., Wiley, 2014.
2. Oracle, *Java SE 17 API Specification*: documentation for `java.lang.Integer`, `java.lang.String`, `java.util.ArrayList`, `java.util.HashMap`, and `java.util.HashSet`.
3. R. Sedgewick, K. Wayne, *Algorithms*, 4th ed., Addison-Wesley, 2011.
4. N. Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall, 1976.

PART I · CHAPTER 2

Algorithm Analysis

A program that solves a problem is often not enough. We need to choose between two programs that do the same job, predict how long ours will take on large inputs, and know when to look for something better. Algorithm analysis answers these questions. We start by timing programs, see where that breaks down, then count primitive operations and introduce Big O, Big Omega and Big Theta. The core of the chapter covers best, average and worst cases, reading complexity off loops, linear and binary search, and two solutions to the prefix averages problem. The last sections deal with memory, correctness and robustness.

Code for this chapter: `github.com/sercankulcu/data-structures-java/Bolum02/src`

LEARNING OUTCOMES

Measuring and counting
Time an algorithm experimentally and analyze it by counting primitive operations.

Asymptotic notation
Express time and space complexity in O, Ω and Θ notation.

From code to complexity
Derive best-, average- and worst-case costs from loop patterns.

Comparison
Compare algorithms for the same problem by cost, correctness and robustness.

2.1 Algorithms and measurement

Every topic in this course raises the same question: how long does this operation take? Searching an array, inserting into a list, deleting from a tree — each data structure makes some operations fast and others slow, and to say which structure suits which job we must give “fast” and “slow” a precise meaning. That is the job of **algorithm analysis**. We begin with the notion of an algorithm and two ways to measure its cost: timing it and counting its operations.

2.1.1 What is an algorithm?

A recipe takes ingredients (the input), works through a sequence of steps, and produces a dish (the output). Its steps are explicit — "boil the water," "add the rice," "simmer for fifteen minutes" — so anyone who follows them gets the same dish, and it ends; nobody stirs forever. An **algorithm** has the same shape: a step-by-step method that takes an input, carries out its steps, and produces an output.

Definition 2.1

An **algorithm** is a method, made up of a finite number of precise and effective steps, that produces the correct output for every valid input of a problem. *Finite* means it eventually stops, *precise* that each step has exactly one meaning, and *effective* that each step can actually be carried out.

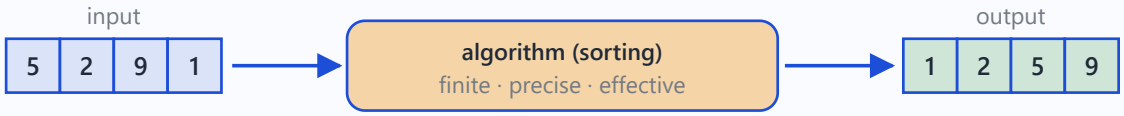


Figure 2.1: An algorithm applies its steps to the input to produce the output. Here a sorting algorithm turns 5, 2, 9, 1 into 1, 2, 5, 9.

What happens inside the box in Figure 2.1 — which numbers are compared and swapped, in what order — differs from algorithm to algorithm; many routes lead from the same input to the same output.

That is where analysis begins: of several algorithms for a problem, some need far less time and memory. On small inputs the difference may go unnoticed, but as the input grows one algorithm may finish in a second while another runs for days; Table 2.7 puts numbers on this gap.

Algorithms are grouped by the kind of problem they solve. Table 2.1 lists three groups that recur throughout the course.

Group	What it does	Examples
Sorting	Puts data in a given order.	bubble sort, merge sort
Searching	Finds a given item in a collection.	linear search, binary search
Graph	Solves problems from graph theory.	shortest path, network flow

Table 2.1: Some common groups of algorithms.

Two searching algorithms, linear and binary search, are studied in detail below (2.3.5–2.3.7). Both answer "Is the target in the array, and at which index?", but one looks at every element and the other at only a few — one of the clearest demonstrations of why the choice of algorithm matters.

Common mistake

Treating an algorithm as the same thing as a program. An algorithm is a method that can be written in Java, in Python, or as pseudocode on paper, and its implementations in different languages have the same analysis. Analysis counts the steps of the method, not features of the language.

2.1.2 What makes an algorithm good

A good algorithm solves its problem effectively, and "effectively" breaks down into four criteria — two about cost, two about reliability:

- **Time complexity**: how the running time, or the number of operations, grows with the input.
- **Space complexity**: how the memory used grows with the input size (2.4.1).
- **Correctness**: the right result for every valid input. A wrong answer is useless, however fast (2.4.2).
- **Robustness**: coping with unexpected inputs and situations without crashing or silently returning a wrong result (2.4.4).

The criteria are not equal. Correctness is non-negotiable: a fast algorithm with wrong answers is always worse than a slow one with right answers. Time and space decide only among correct algorithms. They sometimes conflict — some algorithms buy speed with memory — but not always: in the prefix averages example (2.3.12) the fast solution uses the same memory and simply skips needless work.

Most of this chapter is about time, which most often decides the choice of a data structure. Memory is analyzed with the same notation and methods.

2.1.3 Empirical analysis: System.nanoTime()

The most natural way to learn how long a program takes is to run it and time it — **empirical analysis**. In Java, `System.nanoTime()` returns a number, and the difference between two calls is the elapsed time in nanoseconds (ns, billionths of a second).

The code below times `hasDuplicates`, which checks an array for repeated elements, at several sizes. It compares every pair of elements (code and analysis in 2.3.3); for now we only need to know that on an array without duplicates it examines every pair before returning `false`.

```
for (int n = 1000; n <= 64000; n *= 2) {
  int[] a = new int[n];
  for (int i = 0; i < n; i++) a[i] = i; // no duplicates
  long start = System.nanoTime();
  hasDuplicates(a);                    // the work being timed
  long elapsed = System.nanoTime() - start;  // ns
  System.out.println(n + " " + elapsed / 1e6 + " ms");
}
```

- The outer loop doubles `n` from 1000 — 1000, 2000, 4000, ..., 64000 — so seven sizes are tried.
- `new int[n]` creates an array of `n` cells, and `a[i] = i` fills it with 0, 1, 2, ...; all values differ, so there are no duplicates.
- `start = System.nanoTime()` records the moment just before the call `hasDuplicates(a)`.
- `System.nanoTime() - start` is the elapsed time in nanoseconds; dividing by 10^6 gives milliseconds (ms), printed next to `n`.

The value of `System.nanoTime()` means nothing on its own; it is not the time of day, and only the difference between two readings matters. The time of day — a clock saying "it's 2:05" — comes from `System.currentTimeMillis()`. For short intervals `nanoTime()` is better: its resolution is finer, and it does not jump when the system clock is adjusted.

Good measurements follow three principles:

1. **Measure.** Read `System.nanoTime()` immediately before and after the work. Creating and filling the array stay outside; only the call to `hasDuplicates` is timed.
2. **Repeat.** One measurement is unreliable. The Java Virtual Machine (JVM) interprets code at first and later compiles frequently run parts to machine code; this just-in-time (JIT) compilation slows the first measurements, and the garbage collector and other programs add noise. So we repeat each measurement and take the **median** — the middle value of the sorted measurements, which no single outlier can distort.
3. **Plot.** Plot time against `n`. What matters is not the times themselves but how much the time grows when `n` doubles — the subject of the next subsection.

Table 2.2 shows one run of this code on a laptop. The numbers vary between machines and even between runs; we use them only to show how a measurement is read.

`n`	Time (ms)	Ratio
1,000	8.01	—
2,000	0.86	0.11
4,000	2.71	3.1
8,000	10.64	3.9
16,000	37.83	3.6

`n`	Time (ms)	Ratio
32,000	146.11	3.9
64,000	727.33	5.0

Table 2.2: A sample measurement of `hasDuplicates` (single run). The last column shows how many times longer each row took than the one before.

The first two rows look odd: 1000 elements took about nine times longer than 2000. The culprit is the JIT, not the algorithm — on the first call `hasDuplicates` is still interpreted, by the second it is compiled. The first row is a "warm-up" and is ignored. After that the ratio ranges from 3.1 to 5.0, mostly near 4. The 5.0 in the last row is noise: the large array no longer fits in the processor cache, or another program happened to run. Repeating and taking the median removes most such deviations.

> **Common mistake**
>
> Forgetting to fill the array when measuring the worst case. A Java array created with `new int[n]` is full of zeros, so `hasDuplicates` finds `a[0] == a[1]` on its first comparison and returns `true`. The measurement then shows microseconds even for 64,000 elements and makes the method look very fast. For the worst case, fill the array with distinct values, as `a[i] = i` does.

2.1.4 The doubling experiment

Raw times say little: "727 ms for 64,000 elements" means nothing on another computer. What lasts is how the time *grows with the input*, and the practical way to find it is the **doubling experiment**: double the input size at each step and look at the ratio of consecutive times.

Suppose the running time is roughly $T(n) \approx c \cdot n^k$, where the constant c depends on the machine and the exponent k on the algorithm. Doubling the input gives

$$T(2n) \,/\, T(n) \approx c \cdot (2n)^k \,/\, (c \cdot n^k) = 2^k$$

The constant c cancels, so the ratio is machine-independent: a faster computer shrinks c but not the ratio. A ratio of 2 means $k = 1$, linear growth; 4 means $k = 2$, quadratic; 8 means $k = 3$, cubic.

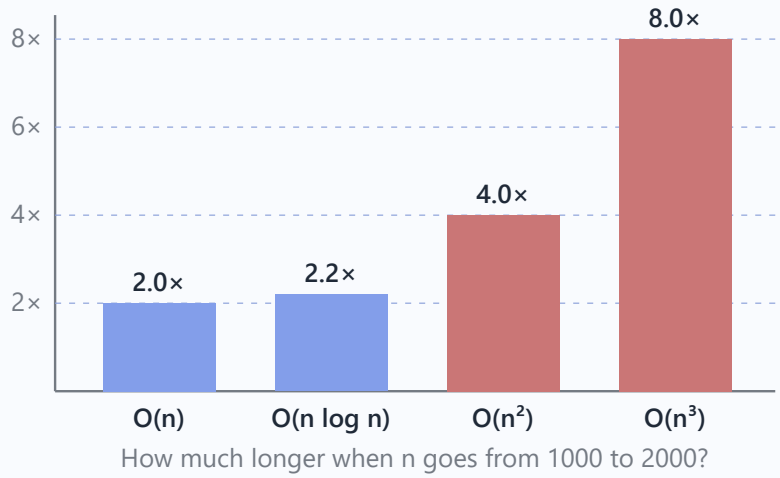


Figure 2.2: Expected time ratios when n goes from 1000 to 2000: 2 for $O(n)$, about 2.2 for $O(n \log n)$, 4 for $O(n^2)$, and 8 for $O(n^3)$.

Figure 2.2 shows the expected ratios for four growth rates. (The $O(\ldots)$ notation is defined in 2.2.1; read it for now as "grows roughly like.") The interesting bar is $n \log n$: the logarithm grows a little when n doubles, so the ratio is slightly above 2:

$$(2000 \cdot \log_2 2000) \,/\, (1000 \cdot \log_2 1000) = 2 \cdot 10.966 \,/\, 9.966 \approx 2.2$$

This is also why a doubling experiment struggles to tell n from $n \log n$: noise easily blurs 2.0 and 2.2.

Example 2.1

Read Table 2.2 as a doubling experiment and predict the time for 128,000 elements.

1. **Warm-up.** Drop the first row (1000 elements); it carries the JIT effect.
2. **Ratios.** The rest are 3.1, 3.9, 3.6, 3.9 and 5.0 — far from both 2 and 8, clustered around 4.
3. **Exponent.** $2^k \approx 4$ gives $k = 2$: the time is roughly $c \cdot n^2$, quadratic.
4. **Prediction.** 128,000 is twice 64,000, and quadratic growth quadruples the time: $4 \cdot 727 \approx 2909$ ms, about 3 seconds. Using exactly 4 keeps the noise out; averaging the measured ratios would carry it into the prediction.

This agrees with the $n(n-1)/2$ comparisons we will derive from the code of `hasDuplicates` (2.3.3), a count that grows like n^2.

Common mistake

Looking at the difference instead of the ratio. From 8,000 to 16,000 the time rises from 10.64 ms to 37.83 ms; the 27 ms difference tells us nothing, while the ratio of about 3.6 does. A doubling experiment always divides, never subtracts.

2.1.5 Limits of empirical analysis

Empirical analysis is concrete — it shows a real program on a real machine — but it has three serious limitations (Goodrich et al., Section 4.1):

- **Results depend on the environment**: the processor, the memory, the JVM version, whatever else is running. Two algorithms must be timed on the same machine under the same conditions; comparing our timing with one from someone else's machine is meaningless.
- **It covers only the inputs tried.** An important kind of input that slows the algorithm down may never be tested. Someone who times `hasDuplicates` only on arrays of zeros never sees its worst case.
- **The algorithm must be fully implemented** before anything can be measured, yet decisions are usually needed before the code exists. Implementing and timing ten methods just to pick one is expensive.

The way past these limits is to *count* an algorithm's steps instead of timing them. Counting is hardware-independent, holds for every input, and needs no running code — pseudocode will do. Empirical analysis does not replace it but is a good check on it; in Example 2.1 measuring and counting agreed.

2.1.6 Counting primitive operations: arrayMax()

First we must decide what to count. A line of Java may become several machine instructions whose timings vary between processors. To avoid such details we assume that certain basic operations each take constant time, and count only how often they run.

Definition 2.2

A **primitive operation** is a basic operation whose running time does not depend on the input size and is bounded by a constant. Assigning to a variable, comparing two values, an arithmetic operation (addition, multiplication, …), accessing an array cell by index, calling a method and returning from one are all primitive operations.

If each primitive operation takes at most some constant time, the running time is at most that constant times the operation count. The constant changes from machine to machine; the count does not — a machine-independent measure.

Our first example, `arrayMax`, finds the largest element of an array:

```
int arrayMax(int[] a) {
  int currentMax = a[0];
  for (int i = 1; i < a.length; i++) {
    if (a[i] > currentMax)
      currentMax = a[i];
  }
  return currentMax;
}
```

It takes the first element as a provisional maximum, scans the rest of the array, and updates `currentMax` whenever an element beats it. When the loop ends, `currentMax` is the largest element of the whole array.

Now count how often each statement runs, for an array of length n, numbering lines from the method header as line 1.

Line	Statement	Times
2	`currentMax = a[0]`	1
3	`i = 1`	1
3	`i < a.length`	n
4	`a[i] > currentMax`	n − 1
5	`currentMax = a[i]`	0 … n − 1
3	`i++`	n − 1
7	`return`	1

Table 2.3: How many times each primitive operation runs in `arrayMax` (n = `a.length`).

- The assignment before the loop (`currentMax = a[0]`) and the initialization `i = 1` run once each.
- The body runs for `i` = 1, 2, …, $n - 1$ — $n - 1$ iterations — so the comparison and `i++` run $n - 1$ times each.
- The condition `i < a.length` is tested once more than the body runs: $n - 1$ times true, then once false at `i = n`, ending the loop — n tests.
- The update `currentMax = a[i]` runs only when the comparison is true, so its count depends on the data: anywhere from never to $n - 1$ times.

Writing u for the number of updates, the total is

$$1 + 1 + n + (n - 1) + u + (n - 1) + 1 = 3n + 1 + u, \quad 0 \le u \le n - 1$$

In the **best case**, `a[0]` is the largest and nothing is updated: $u = 0$, total $3n + 1$. In the **worst case** the array is increasing, every element beats all earlier ones, and every iteration updates: $u = n - 1$, total $3n + 1 + n - 1 = 4n$.

The conclusion matters more than the numbers. Either way the count grows in proportion to n: double n and it roughly doubles. The factors 3 and 4 and the +1 are details that vary between machines anyway, so we drop them and say that `arrayMax` runs in $\Theta(n)$ time in both cases. Θ is defined precisely in 2.2.2.

> **Common mistake**
>
> Forgetting that a loop condition runs once more than the body. In `for (int i = 1; i < n; i++)` the body runs $n - 1$ times but the condition is tested n times, the last test ending the loop. The growth rate is unaffected, but exact-count questions come out wrong.

Which operations count separately is a convention. `currentMax = a[i]` really reads a cell and assigns a value; counting it as two operations turns $3n + 1$ into another linear expression, but the growth stays linear. Changing the counting rules changes only constants — one more reason to drop them.

2.1.7 Code trace: arrayMax(a)

Let us follow the count on `a = {3, 8, 2, 9, 4}`, so $n = 5$. Table 2.4 shows the variables at each step.

Step	`i`	Condition `i < 5`	`a[i] > currentMax`	`currentMax`
start	—	—	—	3
iteration 1	1	true	8 > 3 true	8
iteration 2	2	true	2 > 8 false	8
iteration 3	3	true	9 > 8 true	9
iteration 4	4	true	4 > 9 false	9
exit	5	false	—	returns 9

Table 2.4: Trace of `arrayMax({3, 8, 2, 9, 4})`.

Initially `currentMax = a[0] = 3`. In iteration 1, `a[1] = 8` beats 3 and becomes the maximum; in iteration 2, `a[2] = 2` changes nothing; in iteration 3, `a[3] = 9 > 8` takes over; in iteration 4, `a[4] = 4` is smaller than 9. At `i = 5` the condition fails and the method returns 9.

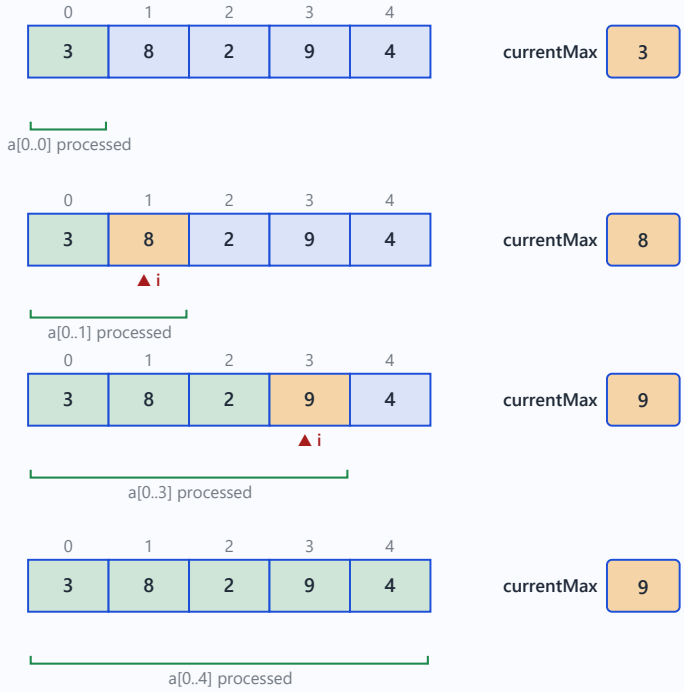


Figure 2.3: Four moments in the trace of `arrayMax` (left to right, top to bottom). (a) `currentMax = a[0] = 3`. (b) Iteration 1 finds 8. (c) Iteration 3 finds 9. (d) The loop has ended; `a[0..4]` is processed and the result is 9. The green line underneath marks the part processed so far.

Checking against the table: the condition was tested 5 times (4 true, 1 false), the comparison made 4 times, `i++` run 4 times, and there were two updates (8 and 9), so $u = 2$. The total, $3 \cdot 5 + 1 + 2 = 18$, lies between the best-case 16 and the worst-case 20.

The green line in Figure 2.3 also hints at correctness: at the start of every iteration, `currentMax` is the largest element of the processed part, and when the loop ends that part is the whole array. In 2.4.3 this becomes a proof by loop invariant.

2.2 Asymptotic notation

For `arrayMax` we found between $3n + 1$ and $4n$ operations and kept only "proportional to n." Every analysis prunes like this, and **asymptotic** notation does it systematically: it looks at what happens as the input grows very large, setting aside details that matter only for small inputs.

2.2.1 Time complexity and Big O

Time complexity expresses an algorithm's running time as a function of the input size n — the number of elements in an array, characters in a string, vertices and edges in a graph, depending on the problem. We rarely need the function's exact form, only how it grows.

Big O gives an *upper bound*. "This algorithm is *O(n)*" means that for large n its running time stays below some constant multiple of n. Big O is usually applied to the worst case, since an upper bound there is a guarantee for every input — but that is a habit, not part of the definition (2.2.5).

Two classes come up constantly:

- ***O(1)***: the operation count does not grow with the input. Reading the first element of an array takes the same steps for three elements or a million. *O(1)* does not mean identical times, only a bound independent of n.
- ***O(n)***: the time grows linearly; doubling n at most roughly doubles it. One pass through an array is *O(n)*.

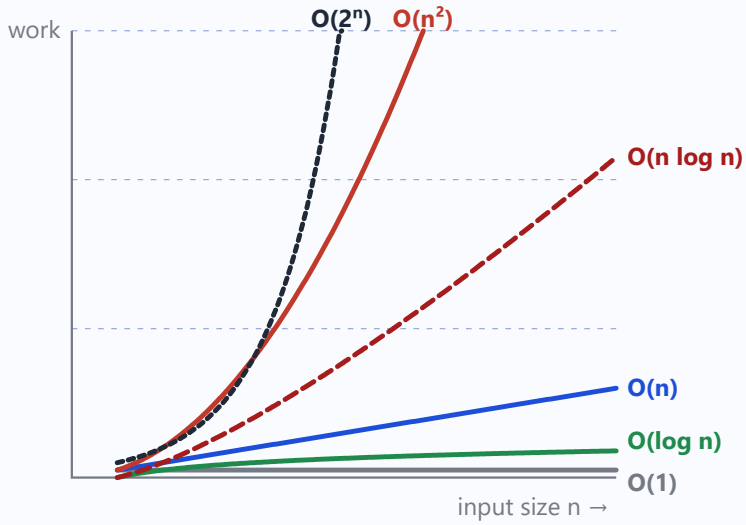


Figure 2.4: How *1*, *log n*, *n*, *n log n*, n^2 and 2^n grow with the input size n ($1 \le n \le 12$, vertical axis cut off at 60). 2^n and n^2 shoot up while *log n* and *1* stay almost flat.

Even with n only up to 12, the differences in Figure 2.4 are plain: 2^n leaves the chart near $n = 6$ and n^2 at $n \approx 7.7$, *n log n* runs clearly above n, and *log n* is only about 3.6 at 12. Table 2.7 shows how far the gaps open on large inputs.

Big O keeps only the *fastest-growing term*, dropping constant factors and slower terms:

$$3n^2 + 5n + 2 \to O(n^2) \cdot 4n + 100 \to O(n) \cdot 7 \to O(1)$$

Why is this safe? In the first example, at $n = 10$ we have $3n^2 = 300$, $5n = 50$ and 2, so the quadratic term is 85% of the total; at $n = 1000$, $3n^2 = 3{,}000{,}000$ against $5n = 5000$, over 99.8%. The smaller terms' share tends to zero. The factor 3 depends on the machine and leaves the growth rate alone: doubling n quadruples both $3n^2$ and n^2.

Common mistake

Assuming $O(1)$ is always fast. The constant can be large: an $O(1)$ operation taking 1000 steps per call is slower than an $O(n)$ operation taking 50 steps at $n = 50$. Asymptotic notation compares large n; on small inputs the constants can decide.

2.2.2 Formal definitions of O, Ω and Θ

Now we make "below a constant multiple for large n" precise. Below, $f(n)$ is the operation count of the algorithm and $g(n)$ a simple comparison function such as n, n^2 or $\log n$.

Definition 2.3

$f(n) = O(g(n))$ if there exist a constant $c > 0$ and an integer $n_0 \geq 1$ such that $f(n) \leq c \cdot g(n)$ for every $n \geq n_0$. Then $g(n)$ is an **upper bound** on $f(n)$: for large enough n, f grows no faster than g.

$f(n) = \Omega(g(n))$ (Big Omega) under the same conditions with the inequality reversed: $f(n) \geq c \cdot g(n)$ for every $n \geq n_0$. Then $g(n)$ is a **lower bound**: f grows no slower than g.

$f(n) = \Theta(g(n))$ (Big Theta) if both $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$; that is, there exist $c_1, c_2 > 0$ and n_0 with $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for every $n \geq n_0$. This is a **tight bound**: f and g grow at the same rate.

Each part has a job. The constant c captures why constant factors don't matter: $c = 3$ closes the gap between $3n$ and n. The threshold n_0 captures why small inputs don't matter. And "for every $n \geq n_0$" demands that the bound hold *everywhere* past the threshold, not at scattered points.

We will apply all three definitions to $f(n) = 2n + 10$.

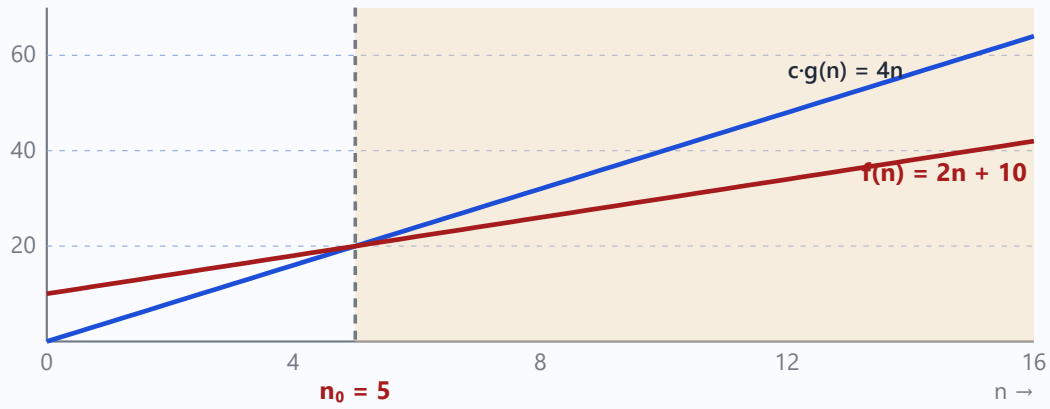


Figure 2.5: Big O. For $n \geq n_0 = 5$ the line $f(n) = 2n + 10$ stays below $4n$, so $f(n) = O(n)$ with $c = 4$. The shaded region is the range $n \geq n_0$ where the inequality is required.

Big O. In Figure 2.5, f starts at 10 and $4n$ at 0, but $4n$ is steeper and catches up where $2n + 10 = 4n$, at $n = 5$. For $n \geq 5$, $2n + 10 \leq 4n$, so $c = 4$ and $n_0 = 5$ show $f(n) = O(n)$. Below 5, f is larger — at $n = 2$, $f = 14$ while $4n = 8$ — but the definition ignores these values.

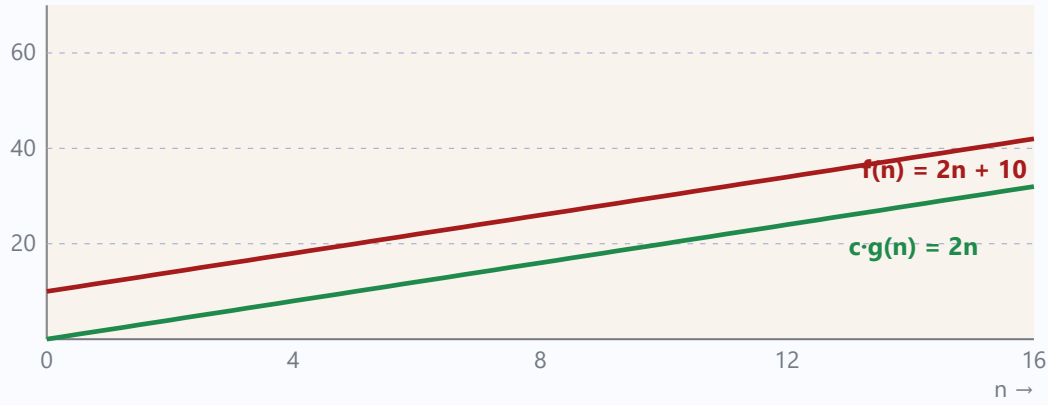


Figure 2.6: Big Omega. $2n + 10 \geq 2n$ for every $n \geq 1$, so $f(n) = \Omega(n)$. The line f lies above $2n$ everywhere.

Big Omega. For a lower bound, $c = 2$ works: since 10 is positive, $2n + 10 \geq 2n$ for every n (Figure 2.6), so $n_0 = 1$ and no threshold is needed. $f(n) = \Omega(n)$: f grows no slower than n.

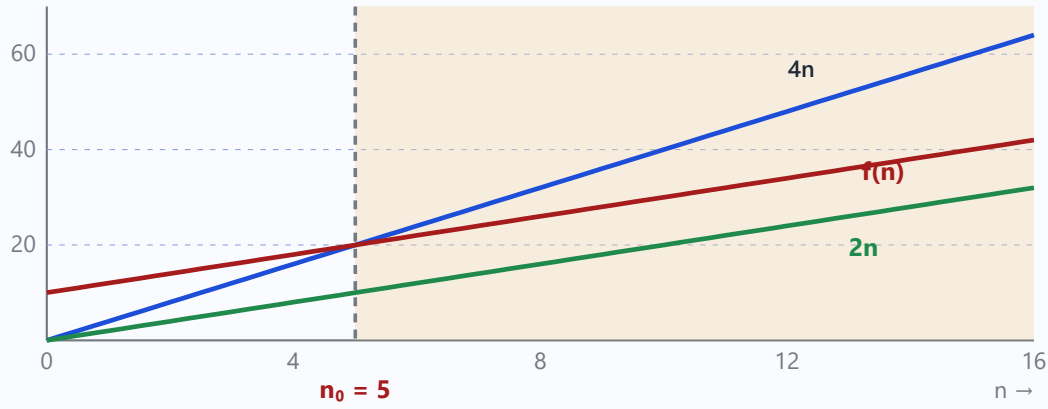


Figure 2.7: Big Theta. For $n \geq 5$, $2n \leq f(n) \leq 4n$; f is squeezed between two lines, so $f(n) = \Theta(n)$.

Big Theta. Together, $2n \leq 2n + 10 \leq 4n$ for $n \geq 5$ (Figure 2.7): f is trapped between lines of slope 2 and 4, so $f(n) = \Theta(n)$ — its growth rate is exactly n.

Think of an exam grade. "At most 90" is an upper bound (O), "at least 70" a lower bound (Ω), and together they narrow the grade down. Asymptotically, "at most" and "at least" hold up to constant factors, and when one function serves as both bounds, the growth rate is known exactly (Θ).

Which to write depends on what we know. O is the most common but can be loose: $2n + 10 = O(n^2)$ is true ($2n + 10 \le n^2$ for $n \ge 5$) yet useless, since it hides the real growth. So we give the tightest bound we can, and when it is known, Θ states it most clearly.

Common mistake

Reading the "=" in $f(n) = O(g(n))$ as true equality. It means "f belongs to the class $O(g)$." $n = O(n^2)$ is true, but $O(n^2) = n$ does not follow. Equality is symmetric; this notation is not.

2.2.3 Using the definition: finding c and n_0

Showing that an expression is in a Big O class means finding a c and n_0. Any pair that works will do. Two approaches are common: bound every term by the largest, or solve the inequality directly.

Example 2.2

Show that $f(n) = 3n^2 + 5n + 2$ is $\Theta(n^2)$.

1. **Upper bound, first approach (enlarge the terms).** For $n \ge 1$, $n \le n^2$ and $1 \le n^2$, so $5n \le 5n^2$ and $2 \le 2n^2$. Adding up, $f(n) \le 3n^2 + 5n^2 + 2n^2 = 10n^2$: $c = 10$, $n_0 = 1$ give $f(n) = O(n^2)$.
2. **Upper bound, second approach (a smaller c).** With $c = 4$, $3n^2 + 5n + 2 \le 4n^2$ holds exactly when $n^2 - 5n - 2 \ge 0$. The positive root of this quadratic is $(5 + \sqrt{33})/2 \approx 5.37$, so it holds for $n \ge 6$. Check: $f(6) = 108 + 30 + 2 = 140 \le 4 \cdot 36 = 144$, but $f(5) = 102 > 100$. So $c = 4$, $n_0 = 6$ is also valid.
3. **Lower bound.** Since $5n + 2$ is positive, $f(n) \ge 3n^2$ for every $n \ge 1$: $c = 3$, $n_0 = 1$ give $f(n) = \Omega(n^2)$.
4. **Conclusion.** $3n^2 \le f(n) \le 10n^2$ for $n \ge 1$, so $f(n) = \Theta(n^2)$.

The approaches show a trade-off: the smaller c, the larger n_0. The choice doesn't matter; the definition only asks for *some* pair.

The first approach works for any polynomial. For $n \ge 1$, every term of a degree-d polynomial with nonnegative coefficients is at most a multiple of n^d, so the sum of the coefficients serves as c; the leading coefficient gives the lower bound.

Theorem 2.1 (Growth rate of polynomials)

Let $f(n) = a_d n^d + a_{d-1} n^{d-1} + \ldots + a_1 n + a_0$ with $a_d > 0$ and all other coefficients nonnegative. Then $f(n) = \Theta(n^d)$, because for $n \ge 1$, $a_d n^d \le f(n) \le (a_d + a_{d-1} + \ldots + a_0) \cdot n^d$.

The theorem also holds with negative coefficients (as long as $a_d > 0$), but n_0 needs more care. For example, $n(n-1)/2 = n^2/2 - n/2$ is $\Theta(n^2)$: for $n \geq 2$, $n/2 \leq n^2/4$, so $n^2/4 \leq n(n-1)/2 \leq n^2/2$. This expression returns in the analysis of `hasDuplicates`.

> **Common mistake**
>
> Letting c depend on n. "Choose $c = n$, so $n^2 \leq c \cdot n$ and $n^2 = O(n)$" is wrong: c is a constant and cannot grow with n. Whatever constant we pick, $n^2 > c \cdot n$ as soon as $n > c$, so n^2 is not $O(n)$.

2.2.4 Rules of thumb

Hunting for c and n_0 every time would be tiresome. A few rules that follow from the definition do most of the work:

- **Drop constant factors.** For a constant $k > 0$, $k \cdot f(n)$ is in the same class as $f(n)$; $5n$ and $n/2$ are both $\Theta(n)$.
- **In a sum, the largest term wins.** If $f_1 = O(g_1)$ and $f_2 = O(g_2)$, then $f_1 + f_2 = O(\max(g_1, g_2))$. Consecutive blocks of code combine this way: $O(n) + O(n^2) = O(n^2)$.
- **In a product, the bounds multiply.** If $f_1 = O(g_1)$ and $f_2 = O(g_2)$, then $f_1 \cdot f_2 = O(g_1 \cdot g_2)$. This gives nested loops: n outer iterations around $\log n$ inner ones cost $O(n \log n)$.
- **The base of a logarithm doesn't matter.** Since $\log_a n = \log_b n / \log_b a$, logarithms to different bases differ by a constant factor — for example, $\log_2 n = \log_{10} n / \log_{10} 2 \approx 3.32 \cdot \log_{10} n$ — so we just write $O(\log n)$.
- **Transitivity.** If $f = O(g)$ and $g = O(h)$, then $f = O(h)$; likewise for Ω and Θ.

For $n = 1{,}000{,}000$, $\log_2 n \approx 19.9$ and $\log_{10} n = 6$, but the ratio is about 3.32 for every n. When n grows a thousandfold, each grows by a fixed *amount*: about 10 for $\log_2$, exactly 3 for $\log_{10}$.

Exponentials are different: changing the base changes the growth rate. $3^n / 2^n = 1.5^n$ grows without bound, so 3^n is not $O(2^n)$. A constant added to the exponent, however, is just a constant factor: $2^{n+3} = 8 \cdot 2^n = \Theta(2^n)$.

> **Common mistake**
>
> Treating $\log_2 n$ and $\log_{10} n$ as different classes. They differ by a factor of about 3.32 and both are $O(\log n)$. This is special to logarithms; for exponentials the base matters, and 2^n and 3^n are in different classes.

2.2.5 Best, average and worst cases

`arrayMax` needed $3n + 1$ operations on a decreasing array and $4n$ on an increasing one. Most algorithms vary far more, so "the running time" must say which input it means.

Definition 2.4

For a fixed input size n:

- The **best case** is the cost of the input of size n that needs the fewest operations.
- The **worst case** is the cost of the input of size n that needs the most operations.
- The **average case** is the expected cost, assuming inputs are drawn from some probability distribution.

Table 2.5 puts the three bounds and the three cases side by side. The top half answers "what kind of bound?", the bottom half "for which input?" — independent questions.

Concept	What it says / asks	Linear search
$O(g(n))$: upper bound	For large enough n, the work grows no faster than $g(n)$.	—
$\Omega(g(n))$: lower bound	For large enough n, the work grows no slower than $g(n)$.	—
$\Theta(g(n))$: tight bound	Upper and lower bounds coincide; the growth is in the class of $g(n)$.	—
Best case	How many operations does the most favorable valid input need?	$\Theta(1)$ if the first element is the target
Average case	What is the expected cost under an assumed input distribution?	$\Theta(n)$ if all target positions are equally likely
Worst case	What maximum cost is guaranteed for every valid input?	$\Theta(n)$ if the target is last or absent

Table 2.5: Asymptotic bounds and case analyses, with linear search as the example in the right column.

Linear search checks the elements one by one from the front and stops at the target (code in 2.3.5). Let us work out the three cases for `[4, 8, 15, 16, 23, 42]`, $n = 6$.

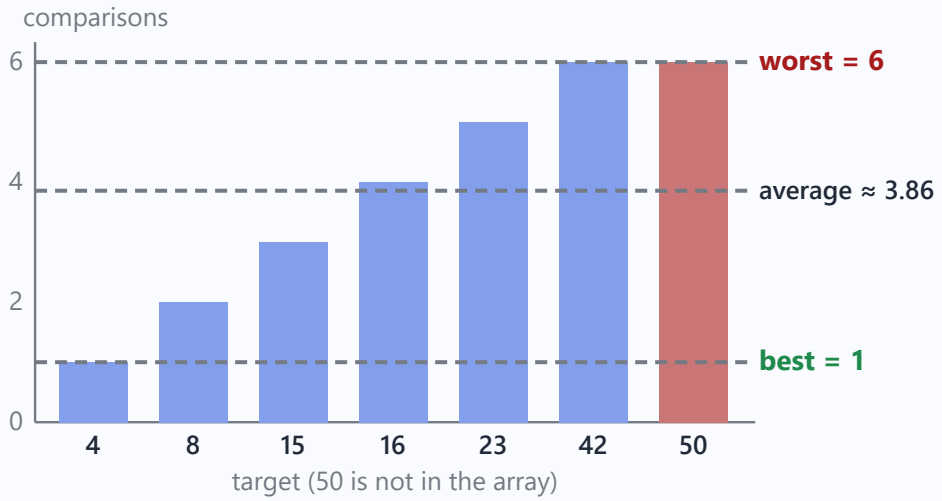


Figure 2.8: Comparisons made by linear search for each possible target in `[4, 8, 15, 16, 23, 42]`; 50 is not in the array. The best case is 1 comparison and the worst 6; with the seven targets equally likely, the average is about 3.86.

Example 2.3

1. **Cost of each target.** 4 is found on the first comparison, 8 on the 2nd, 15 on the 3rd, 16 on the 4th, 23 on the 5th and 42 on the 6th. For 50, which is absent, all six elements are checked: 6 comparisons.
2. **Best case.** 1 comparison, when the target is first. This does not depend on n: $\Theta(1)$.
3. **Worst case.** 6, that is, n, when the target is last or absent: $\Theta(n)$.
4. **Average case.** If the seven possible targets (six elements and one missing value) are equally likely, the average is $(1 + 2 + 3 + 4 + 5 + 6 + 6) / 7 = 27 / 7 \approx 3.86$.

Another assumption gives another average. If the target is always present and the six positions are equally likely, it is $(1 + 2 + \ldots + 6)/6 = 21/6 = 3.5$; in general, with n equally likely positions, $(1 + 2 + \ldots + n)/n = (n + 1)/2$. That is still $\Theta(n)$: we look at about half the array, and half is proportional to n.

Which case to report depends on the purpose:

- The **worst case** is a guarantee. “Never more than $4n$ steps on any input” is reliable information for a system designer, and usually easy to compute — hence the most often reported.
- The **average case** describes what to expect in practice, but it assumes a probability distribution, misleads when the assumption is wrong, and is often harder to compute.
- The **best case** alone is misleading. Almost any search is fast when the target happens to sit where it looks first, which says next to nothing about the algorithm in general.

Common mistake

Thinking that O means the worst case and Ω the best case. O, Ω and Θ bound the growth of a function; best, average and worst case specify whose cost we study. The ideas are independent, and any notation can describe any case. Linear search has a $\Theta(1)$ best case and a $\Theta(n)$ worst case, and its running time overall is $O(n)$ and $\Omega(1)$.

2.2.6 Amortized analysis

In some data structures most operations are cheap but an expensive one comes along now and then, and worst-case analysis paints too gloomy a picture. **Amortized analysis** divides the total cost of a long sequence of operations by the number of operations.

The standard example is an array that grows when full. Java's `ArrayList` keeps its elements in an array; adding at the end usually means writing into an empty cell, but when the array is full a new one twice as large is allocated and every element copied over — an $O(n)$ addition.

Example 2.4

Add 16 elements to an empty growable array of capacity 1 that doubles when full.

1. **Resizes.** The array fills at capacities 1, 2, 4 and 8, so the 2nd, 3rd, 5th and 9th additions resize, copying 1, 2, 4 and 8 elements. The 16th addition fills the final capacity of 16 exactly without a resize.
2. **Total work.** 16 writes plus $1 + 2 + 4 + 8 = 15$ copies: 31 steps.
3. **Per addition.** $31 / 16 \approx 1.94$ steps. One addition (the 9th) copied 8 elements, yet the average stays below 2.

In general, n additions copy $1 + 2 + 4 + \ldots$ elements; the last term is below n, so the sum is below $2n$, and with the writes the total is below $3n$. The cost per addition is constant: adding at the end is **amortized $O(1)$**.

This is not average-case analysis, which computes an expected cost under a probability distribution. No probability is involved: for every sequence of operations, the worst included, the total is below $3n$ — a guarantee for the sequence rather than for each operation.

Common mistake

Reading amortized $O(1)$ as "every operation is $O(1)$." An addition that triggers a resize still takes $O(n)$, and in a real-time system — a controller whose every operation must meet a deadline, say — that occasional delay can matter. An amortized bound speaks only about totals over long sequences.

2.2.7 Big O classes

Most analyses land in one of a few standard classes, listed in Table 2.6 from slowest- to fastest-growing.

Notation	Name	Example	Why?
$O(1)$	constant	accessing the first element of an array	same access time whatever the array size
$O(\log n)$	logarithmic	binary search in a sorted array	the search space halves at each step
$O(n)$	linear	scanning an array from start to end	each element is processed once
$O(n \log n)$	linearithmic	merge sort	divide and conquer: $\log n$ levels, n work per level
$O(n^2)$	quadratic	comparing all pairs of elements	two nested loops
$O(2^n)$	exponential	generating all subsets	a set of n elements has 2^n subsets

Table 2.6: Common Big O classes.

$O(\log n)$ algorithms shrink the problem by a constant fraction per step, finishing in about 20 steps even for a million elements. $O(n)$ algorithms look at each element a constant number of times; since reading the input alone takes n steps, many problems allow nothing better. $O(n \log n)$ is the class of good sorting algorithms, $O(n^2)$ the natural result of comparing every element with every other, and $O(2^n)$ the price of trying every possibility — each increase of n by one doubles the time.

Merge sort's "$\log n$ levels" deserve a word. It splits the array in two, sorts each half the same way, and merges. An array of n elements can be halved $\log_2 n$ times before the pieces are single elements, and the merges on each level touch n elements in total: $\log n$ levels times n work is $n \log n$.

2.2.8 What happens as n grows?

Numbers show the gaps best. Table 2.7 gives each function at three values of n.

Complexity	$n = 10$	$n = 100$	$n = 1{,}000$
$\log_2 n$	3.3	6.6	10
n	10	100	1,000
$n \log_2 n$	33	664	9,966
n^2	100	10,000	1,000,000
2^n	1,024	$\approx 1.3 \cdot 10^{30}$	$\approx 1.1 \cdot 10^{301}$

Table 2.7: Values of growth functions (rounded).

Two things stand out. $\log_2 n$ barely moves: a hundredfold increase in n takes it only from 3.3 to 10. And 2^n, still reasonable at $n = 10$ (1,024), is a 31-digit number at $n = 100$.

Example 2.5

How long does a computer doing 10^9 operations per second need for a 2^n-step algorithm with $n = 100$?

1. **Steps.** $2^{100} \approx 1.27 \cdot 10^{30}$.
2. **Seconds.** $1.27 \cdot 10^{30} / 10^9 = 1.27 \cdot 10^{21}$ seconds.
3. **Years.** A year is about $365.25 \cdot 24 \cdot 3600 \approx 3.16 \cdot 10^7$ seconds, so $1.27 \cdot 10^{21} / 3.16 \cdot 10^7 \approx 4 \cdot 10^{13}$ years.

The universe is about $1.4 \cdot 10^{10}$ years old, so this is some three thousand times its age. The same computer does $n^2 = 10{,}000$ steps in 10 microseconds.

Faster hardware cannot rescue an exponential algorithm: a computer a thousand times faster would still need $4 \cdot 10^{10}$ years. A thousandfold speedup lets a 2^n algorithm handle an n only 10 larger in the same time ($2^{10} \approx 1000$), but a quadratic one an n about 32 times larger ($\sqrt{1000} \approx 31.6$). On large inputs the choice of algorithm matters far more than the hardware.

2.3 From code to complexity

Now for the real task: reading a piece of code and stating its complexity. We seldom need to count every operation; what matters is how often the loops run. Loop-free code is $O(1)$, and loops fall into a handful of patterns. We introduce the patterns, then apply them to sum, linear and binary search, duplicate checking and prefix averages.

2.3.1 Loop patterns

A loop costs the number of iterations times the cost of one pass through the body; with a constant-cost body, that is proportional to the iteration count. Table 2.8 lists the common patterns, their counts for $n = 16$, and their classes.

Pattern	Body runs	$n = 16$	Class
`for (i = 0; i < n; i++)`	n	16	$O(n)$
`for (i = 0; i < n; i += 2)`	$n / 2$	8	$O(n)$
`for (i = 1; i < n; i *= 2)`	$\lceil \log_2 n \rceil$	4	$O(\log n)$
nested: `i < n`, `j < n`	n^2	256	$O(n^2)$
nested: `i < n`, `j < i`	$n(n-1)/2$	120	$O(n^2)$
two consecutive `i < n` loops	$2n$	32	$O(n)$

Table 2.8: Loop patterns and how many times the body runs.

Simple loop. `i` goes from 0 up by 1 and stops at `n`: values 0, 1, ..., $n - 1$, so n iterations.

Step of 2. `i` takes 0, 2, 4, ...: $n/2$ iterations for even n, $\lceil n/2 \rceil$ for odd. With a step of 2 or 100, the count is a fixed fraction of n and the loop is $O(n)$; a bigger step speeds it up without changing its class.

Doubling loop. `i` doubles — 1, 2, 4, 8, ... — while below n. After k iterations $i = 2^k$, so the loop stops the first time $2^k \geq n$, at $k = \lceil \log_2 n \rceil$. For $n = 16$, `i` is 1, 2, 4, 8 and stops at 16: 4 iterations (traced in 2.3.2). A loop that starts `i` at n and halves it behaves the same way.

Nested loops. The inner loop runs in full on every outer iteration. If it runs n times regardless of the outer loop, the total is $n \cdot n = n^2$. If its bound depends on the outer variable (`j < i`), we add instead of multiplying: 0 iterations at `i = 0`, 1 at `i = 1`, ..., $n - 1$ at `i = n - 1`, for a total of

$$0 + 1 + 2 + \ldots + (n - 1) = n(n - 1)/2$$

This formula is easy to remember: write the sum forward and backward and add the rows term by term — $0 + (n - 1)$, $1 + (n - 2)$, ... — giving n pairs of $n - 1$, which is twice the sum. For $n = 16$, $16 \cdot 15 / 2 = 120$. The inner loop averages half the outer count, but the factor 1/2 is dropped and the result is still $O(n^2)$.

Consecutive loops. One loop after another adds costs: $n + n = 2n$, which is $O(n)$ — far cheaper than nesting.

Two general rules follow:

- **Consecutive blocks add, and the largest dominates:** $O(n) + O(n^2) = O(n^2)$. A linear scan followed by a nested loop costs what the nested loop costs.
- **Nested loops multiply:** outer iterations × inner iterations — or, if the inner count depends on the outer variable, the sum over all iterations.

Example 2.6

What is `s` after this code runs, and what is its complexity? ($n = 16$)

```
for (i = 0; i < n; i++) s++;
for (i = 0; i < n; i++) for (j = 1; j < n; j *= 2) s++;
```

1. **First block.** A simple loop: 16 iterations, `s` becomes 16. $O(n)$.
2. **Second block, inner loop.** `j` is 1, 2, 4, 8: $\lceil \log_2 16 \rceil = 4$ iterations, independent of the outer variable.
3. **Second block, total.** 16 outer iterations of 4 inner ones: $16 \cdot 4 = 64$. $O(n \log n)$.
4. **Whole fragment.** `s = 16 + 64 = 80`, and $O(n) + O(n \log n) = O(n \log n)$.

Common mistake

Memorizing "two nested loops means n^2." That holds only when both loops run in proportion to n. With an inner `j *= 2` the total is $n \log n$; with an inner loop of fixed length (say 10) it is $O(n)$. Find each loop's count separately, then multiply.

2.3.2 Code trace: countSteps(16)

Here is a doubling loop inside a method. `countSteps` counts how many doublings take 1 up to n:

```
int countSteps(int n) {
  int k = 0;
  for (int i = 1; i < n; i *= 2)
    k++;
  return k;
}
```

The counter `k` starts at 0, `i` starts at 1 and doubles via `i *= 2`, and each iteration increments `k`, so at the end `k` is the number of iterations.

Iteration	`i` at start	`i < 16` ?	`k` (after)
1	1	true	1
2	2	true	2
3	4	true	3
4	8	true	4
—	16	false, loop ends	returns 4

Table 2.9: Trace of `countSteps(16)`.

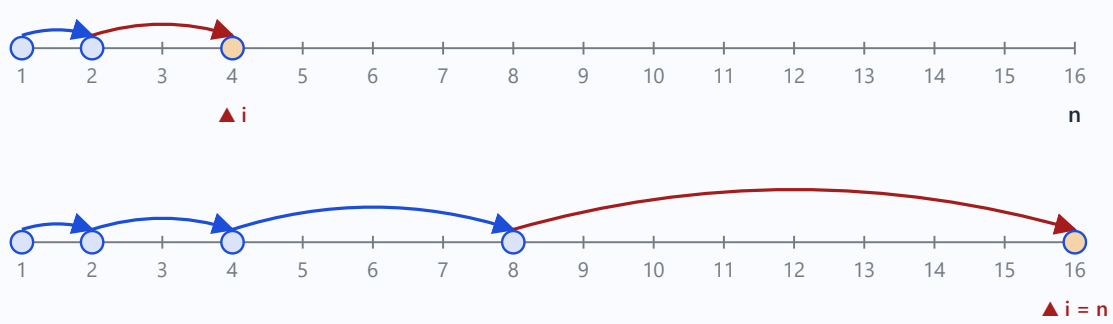


Figure 2.9: `countSteps(16)` on a number line. Top: the third iteration, after `i` has jumped from 1 to 2 and from 2 to 4. Bottom: the final state; `i` jumps from 8 to 16 and the loop ends because `i = n`. Each jump is twice as long as the one before.

Figure 2.9 shows why the loop runs so few times: each jump doubles the previous one — 1, 2, 4, 8 — so four jumps reach 16, skipping most of the numbers on the way.

The result, 4, is $log_2\ 16$; in general `countSteps(n)` returns $\lfloor log_2\ n \rfloor$. It grows very slowly: `countSteps(17) = 5`, `countSteps(1000) = 10`, `countSteps(16000) = 14`. A thousandfold larger n adds only about 10 iterations, because $2^{10} = 1024 \approx 1000$. The method is $O(\log n)$.

> **Common mistake**
>
> Starting a doubling loop at 0. In `for (int i = 0; i < n; i *= 2)`, `i` stays 0 because $0 \cdot 2 = 0$, the condition stays true, and the loop never ends. Doubling loops start at 1; halving loops must stop before reaching 0.

2.3.3 Three methods, three classes

Three short methods, each taking an `int` array and differing only in loop structure:

$O(1)$: the first element.

```
int first(int[] a) {
    return a[0];
}
```

No loop: one array access and a return. Indexing computes the address directly — start address plus index times element size — regardless of the array's length. The method is $O(1)$.

$O(n)$: the sum.

```
int sum(int[] a) {
    int s = 0;
    for (int x : a)  // n times
        s += x;
    return s;
}
```

`for (int x : a)` is the enhanced `for` (for-each) loop: `x` takes each element in turn. The body — one addition, one assignment — runs n times; the code outside the loop is constant. The method is $O(n)$.

$O(n^2)$: checking for duplicates.

```
boolean hasDuplicates(int[] a) {
    for (int i = 0; i < a.length; i++)
        for (int j = i + 1; j < a.length; j++)
            if (a[i] == a[j])
                return true;
    return false;
}
// outer loop n, inner loop ≈ n/2
// → n(n-1)/2 comparisons
```

For each index `i`, the inner loop compares `a[i]` with every element to its right, starting at `j = i + 1`. That start ensures no element is compared with itself and each pair only once (after `a[0]` vs `a[3]`, there is no `a[3]` vs `a[0]`). The method returns `true` at the first equal pair, `false` if there is none.

In the worst case, with no duplicates, the inner loop runs $n - 1$ times for `i = 0`, $n - 2$ for `i = 1`, ..., once for `i = n - 2`, and not at all for `i = n - 1`:

$$(n - 1) + (n - 2) + \ldots + 1 + 0 = n(n - 1)/2 = n^2/2 - n/2$$

Dropping constants and the smaller term gives $O(n^2)$: about $n/2$ inner iterations times n outer ones is still quadratic. This is also the number of distinct pairs among n elements, so any algorithm that examines every pair makes at least this many comparisons.

The best case finds a match at once: if `a[0] == a[1]`, one comparison suffices, $\Theta(1)$ — exactly the array of zeros in 2.1.3.

2.3.4 Code trace: sum()

Tracing `sum` together with its caller also shows how a method call proceeds.

```
int sum(int[] a) {
    int s = 0;
    for (int x : a) {
        s += x;
    }
    return s;
}

// in main:
System.out.println(sum(a));
```

Let `a = {7, 2, 9}` in `main`. Calling `sum(a)` pushes a new **stack frame** holding the parameters and locals (`s`, `x`). The parameter `a` refers to *the same array* as in `main`: a Java array variable is a **reference** holding the array's location, and only the location is passed, so the call costs the same for

any array size.

Step	`x`	`s`	Notes
call	—	—	new frame pushed; `a` refers to the same array
`s = 0`	—	0	the sum is initialized
iteration 1	7	7	*0 + 7*
iteration 2	2	9	*7 + 2*
iteration 3	9	18	*9 + 9*
loop ends	—	18	no elements left
`return`	—	—	frame popped; 18 goes back to the caller and `println` prints 18

Table 2.10: Trace of `sum({7, 2, 9})`.

The body ran $n = 3$ times. Pushing the frame, initializing `s`, returning and printing do not depend on the array size; with 3 million elements only the iteration count would change. The time has the form $c_1 \cdot n + c_2$: $O(n)$.

> **Common mistake**
>
> Assuming that passing an array copies it and takes $O(n)$. Java passes the reference, so the call is $O(1)$. As a consequence, `a[0] = 100` inside the method also changes the array in `main` — they are the same array.

2.3.5 Linear search

Linear search, the simplest way to find a value, examines the elements in order from the front.

```
int linearSearch(int[] a, int target) {
    for (int i = 0; i < a.length; i++)
        if (a[i] == target) return i;
    return -1;
}
```

Each iteration compares `a[i]` with the target; on a match the method returns `i` at once. If the loop finishes, the target is absent and the method returns `-1`, which, not being a valid index, means “not found.”

We trace it on a sorted array, `a = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91}` ($n = 10$), with target 23.

`i`	`a[i]`	`a[i] == 23` ?	Result
0	2	no	`i` advances
1	5	no	`i` advances

`i`	`a[i]`	`a[i] == 23` ?	Result
2	8	no	`i` advances
3	12	no	`i` advances
4	16	no	`i` advances
5	23	**yes**	`return 5`

Table 2.11: Trace of `linearSearch(a, 23)`.

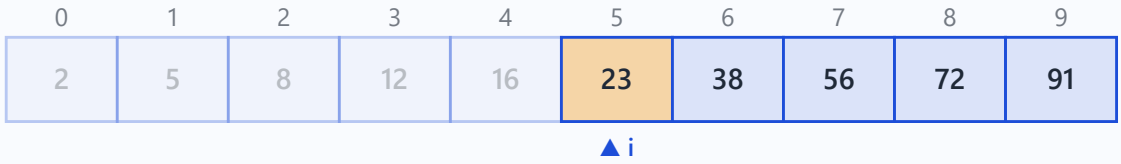


Figure 2.10: The last step of linear search. The faded cells (0–4) were ruled out one at a time; 23 is found at index 5. Each step rules out just one element.

The target took 6 comparisons. Linear search ignores the fact that the array is sorted: learning that 2 is less than 23 says nothing about the other elements, so each step rules out one element (Figure 2.10). A target that is last or absent (say 40) would cost all 10 comparisons.

The analysis matches Example 2.3: best case (target first) 1 comparison, $\Theta(1)$; worst case (last or absent) n, $\Theta(n)$; average with equally likely positions $(n + 1)/2$, also $\Theta(n)$. With only a few variables, the extra space is $O(1)$.

Linear search's strength is that it needs no sorting, and on an unsorted array nothing is faster: we cannot rule out an element without looking at it, so the worst case must examine them all.

2.3.6 Binary search

A sorted array allows much better. Nobody searches a phone book from page one: we open it in the middle, set aside the half that cannot contain the name, and repeat on the rest. **Binary search** is this method as an algorithm.

```
int binarySearch(int[] a, int target) {
    int lo = 0, hi = a.length - 1;
    while (lo <= hi) {
        int mid = (lo + hi) / 2;
        if (a[mid] == target) return mid;
        if (a[mid] < target) lo = mid + 1;
        else hi = mid - 1;
    }
    return -1;
}
```

- `lo` and `hi` bound the range that could hold the target, initially the whole array: `lo = 0`, `hi = a.length - 1`.
- `while (lo <= hi)`: the search continues while the range is nonempty; `lo > hi` means it is empty.
- `mid = (lo + hi) / 2` is the middle index. Integer division in Java discards the fraction: *(0 + 9)/2 = 4*.
- If `a[mid] == target`, `mid` is returned.
- If `a[mid] < target`, the sorted order puts the target right of `mid`: `lo = mid + 1` discards the left half, middle included.
- Otherwise `a[mid] > target` and the target can only be to the left: `hi = mid - 1`.
- If the loop ends, `-1` is returned.

The power lies in one comparison discarding half the range. In linear search, `a[i] ≠ target` rules out one element; in binary search, `a[mid] < target` tells us, thanks to the sorted order, that everything left of `mid` is smaller too.

Same array, same target: `a = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91}`, target 23.

Iteration	`lo`	`hi`	`mid`	`a[mid]`	Decision
1	0	9	4	16	16 < 23 → `lo = 5`
2	5	9	7	56	56 > 23 → `hi = 6`
3	5	6	5	23	equal → `return 5`

Table 2.12: Trace of `binarySearch(a, 23)`.



Figure 2.11: The three iterations of `binarySearch(a, 23)` (top to bottom). Iteration 1: range 0–9, `mid = 4`. Iteration 2: the left half is gone, range 5–9, `mid = 7`. Iteration 3: the right side is gone too, range 5–6, `mid =`

`5`, and 23 is found. Faded cells have been ruled out.

In iteration 1, `mid = (0 + 9)/2 = 4` and `a[4] = 16` is less than 23, so `lo = 5`: five elements gone in one comparison. In iteration 2, `mid = (5 + 9)/2 = 7` and `a[7] = 56` is too large, so `hi = 6`. In iteration 3, `mid = (5 + 6)/2 = 5` and `a[5] = 23`: found.

Three iterations against linear search's six comparisons, the range halving each time — 10, 5, 2. A twofold gain on this small array becomes dramatic on large ones.

Common mistake

Using binary search on an unsorted array. Concluding from `a[mid] < target` that everything to the left is smaller is valid only for sorted data. On unsorted data the method raises no error but may return `-1` for a value that is present. In `{5, 1, 3}`, 1 is found at once at `mid = 1`, but a search for 5 sees `a[1] = 1 < 5`, discards the left half, and never finds it.

2.3.7 The worst case of binary search

The worst case is an absent target, which shrinks the range all the way down. Let us search the same array for 40, which would lie between 38 and 56.

Iteration	`lo`	`hi`	`mid`	`a[mid]`	Decision
1	0	9	4	16	16 < 40 → `lo = 5`
2	5	9	7	56	56 > 40 → `hi = 6`
3	5	6	5	23	23 < 40 → `lo = 6`
4	6	6	6	38	38 < 40 → `lo = 7`
—	7	6	—	—	`7 ≤ 6` false → `return -1`

Table 2.13: Trace of `binarySearch(a, 40)`; the target is not in the array.



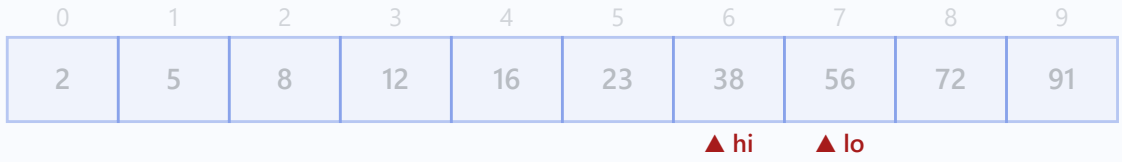


Figure 2.12: The end of `binarySearch(a, 40)`. Top: iteration 4, the range down to one element (index 6, value 38). Bottom: after the loop, `lo = 7` and `hi = 6`; `lo` has passed `hi` and the range is empty.

The first two iterations match the search for 23. In iteration 3, `a[5] = 23` is too small, so `lo = 6`, leaving only 38 at index 6. In iteration 4, 38 is too small as well, so `lo = 7`. Now `lo = 7 > hi = 6` (Figure 2.12), the loop ends, and the method returns `-1`.

As a bonus, the final `lo` is where the target would be inserted: 40 belongs between 38 (index 6) and 56 (index 7), at index 7. Some algorithms for inserting into sorted arrays exploit this.

Even the worst case took just 4 iterations, against 10 comparisons for linear search. The following theorem explains why.

Theorem 2.2 (Iterations of binary search)

On a sorted array of $n \geq 1$ elements, `binarySearch` makes at most $\lfloor \log_2 n \rfloor + 1$ iterations. Binary search therefore runs in $O(\log n)$ time.

Let the range hold m elements. An unsuccessful iteration keeps only the part left or right of `mid`, and discards the middle too, so at most $\lfloor m/2 \rfloor$ elements remain. Starting from n and at least halving each time, at most $\lfloor n/2^k \rfloor$ remain after k iterations — zero once $2^k > n$. So after $k = \lfloor \log_2 n \rfloor + 1$ iterations the range is certainly empty.

For $n = 10$, $\lfloor \log_2 10 \rfloor + 1 = 3 + 1 = 4$, and the counts went $10 \to 5 \to 2 \to 1 \to 0$. Table 2.14 compares the worst cases of the two searches for several sizes.

n	Linear search	Binary search ($\lfloor \log_2 n \rfloor + 1$)
10	10	4
1,000	1,000	10
1,024	1,024	11
1,000,000	1,000,000	20

Table 2.14: Worst-case comparisons for linear search and iterations for binary search.

For a million sorted elements, binary search needs at most 20 iterations where linear search may need a million comparisons — a factor of fifty thousand that widens as n grows. The price is keeping the array sorted, easily paid when it is sorted once and searched many times.

Common mistake

Computing the middle as `(lo + hi) / 2`, which overflows on very large arrays. A Java `int` holds at most $2^{31} - 1 \approx 2.1 \cdot 10^9$, so with `lo = 1,500,000,000` and `hi = 2,000,000,000`, `lo + hi = 3.5 · 10^9` is too large. Java raises no error: the sum silently wraps to a negative number, `(lo + hi) / 2` becomes `-397483648`, and `a[mid]` throws `ArrayIndexOutOfBoundsException`. The safe form is `lo + (hi - lo) / 2`; `hi - lo` never overflows, and the result is the correct `1750000000`. This bug went unnoticed for years in the Java library's own binary search (Bloch, 2006).

2.3.8 Code trace: hasDuplicates()

Now we trace `hasDuplicates` from 2.3.3 on `a = {4, 7, 1, 7}`, where 7 appears twice.

```
boolean hasDuplicates(int[] a) {
    for (int i = 0; i < a.length; i++)
        for (int j = i + 1; j < a.length; j++)
            if (a[i] == a[j])
                return true;
    return false;
}
```

Comparison	i	j	a[i] vs a[j]	Result
1	0	1	4 vs 7	different
2	0	2	4 vs 1	different
3	0	3	4 vs 7	different
4	1	2	7 vs 1	different
5	1	3	7 vs 7	**equal** → `return true`

Table 2.15: Trace of `hasDuplicates({4, 7, 1, 7})`.



Figure 2.13: The first and last comparisons in the trace of `hasDuplicates`. Left, `i = 0`, `j = 1`: 4 is compared with 7. Right, `i = 1`, `j = 3`: the two 7s are equal and the method returns `true`.

With `i = 0`, `a[0] = 4` is compared with the three elements to its right, none equal. With `i = 1`, the inner loop starts at `j = 2`: `a[1] = 7` is compared with 1, then with `a[3] = 7` — a match, so the method returns `true` on the fifth comparison, and the `i = 2` iteration never runs.

Without duplicates — for `{4, 7, 1, 9}`, say — all pairs would be compared: $n(n-1)/2 = 4 \cdot 3 / 2 = 6$ comparisons. This $O(n^2)$ worst case is why the times in Table 2.2 roughly quadrupled with each doubling.

We can do better: after sorting, equal elements are neighbors, so $O(n \log n)$ for the sort plus an $O(n)$ scan suffices. The next example turns a quadratic algorithm into a linear one another way: by cutting redundant work.

2.3.9 The prefix averages problem

How different can two algorithms for the same problem be in cost? A concrete problem answers that (Goodrich et al., Section 4.3).

Definition 2.5

The **prefix averages** problem: given an array `x[0..n-1]` of n numbers, compute the array `a` whose entry for each j is the average of the first $j + 1$ elements:

$$a[j] = (x[0] + x[1] + \ldots + x[j]) / (j + 1), \quad j = 0, 1, \ldots, n - 1$$

A "prefix" is a part that starts at the beginning, `x[0..j]`, and we want every prefix's average. Examples abound: a stock's average daily price since January, a student's grade average since the start of term. Moving averages and financial trend analysis rest on similar calculations.

Example 2.7

Compute the prefix averages of `x = [4, 8, 6, 2]`.

1. $a[0] = 4 / 1 = 4.0$
2. $a[1] = (4 + 8) / 2 = 12 / 2 = 6.0$
3. $a[2] = (4 + 8 + 6) / 3 = 18 / 3 = 6.0$
4. $a[3] = (4 + 8 + 6 + 2) / 4 = 20 / 4 = 5.0$

Result: `a = [4.0, 6.0, 6.0, 5.0]`.

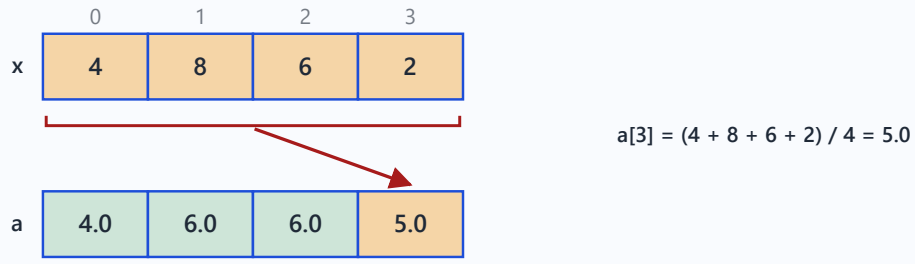


Figure 2.14: The last step of prefix averages: `a[3]` is the average of `x[0..3]`, *(4 + 8 + 6 + 2)/4 = 5.0*. The green cells were computed in earlier steps.

There is a repetition here: 18, the numerator of `a[2]`, is just 12, the numerator of `a[1]`, plus `x[2] = 6`. The two solutions differ on exactly this point. The first recomputes each sum from scratch and takes $O(n^2)$; the second carries the sum forward and takes $O(n)$.

2.3.10 prefixAverage1: summing from scratch

```
double[] prefixAverage1(double[] x) {
  int n = x.length;
  double[] a = new double[n];
  for (int j = 0; j < n; j++) {
    double total = 0;
    for (int i = 0; i <= j; i++)
      total += x[i];
    a[j] = total / (j + 1);
  }
  return a;
}
```

- `a = new double[n]` allocates the result array, which Java fills with 0.0.
- The outer loop computes one prefix average per `j`.
- `double total = 0` sits inside the outer loop, so the sum restarts from zero for every `j`.
- The inner loop adds `x[0]` through `x[j]` to `total`. The condition is `i <= j`; with `<`, `x[j]` would be left out.
- `a[j] = total / (j + 1)` stores the average; `total` is a `double`, so the fraction is kept.

Table 2.16 shows every inner iteration for `x = [4, 8, 6, 2]`.

`j`	Elements added	`total` (intermediate values)	Additions	`a[j]`
0	x[0]	0 → 4.0	1	4.0 / 1 = 4.0
1	x[0], x[1]	0 → 4.0 → 12.0	2	12.0 / 2 = 6.0
2	x[0] … x[2]	0 → 4.0 → 12.0 → 18.0	3	18.0 / 3 = 6.0

`j`	Elements added	`total` (intermediate values)	Additions	`a[j]`
3	x[0] … x[3]	0 → 4.0 → 12.0 → 18.0 → 20.0	4	20.0 / 4 = 5.0

Table 2.16: Trace of `prefixAverage1([4, 8, 6, 2])`.

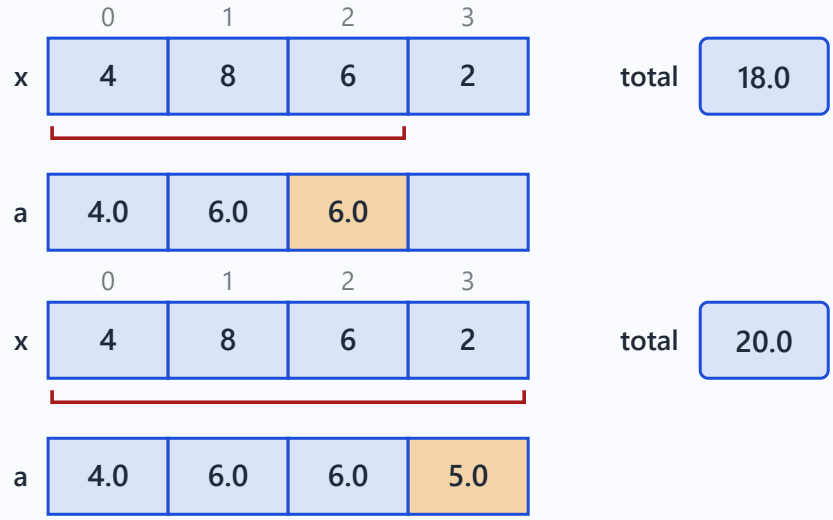


Figure 2.15: Two moments from the trace of `prefixAverage1`. Top: `j = 2` has finished, `total = 18.0`, `a[2] = 6.0`. Bottom: `j = 3` has finished; `total` restarted from zero, re-added all four elements and reached 20.0, so `a[3] = 5.0`.

The third column exposes the weakness: `x[0] = 4` is added four times, `x[1] = 8` three times, `x[2] = 6` twice. Each iteration forgets the previous sum and redoes the same additions.

Iteration `j` runs the inner loop *j + 1* times, so the total number of additions is

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

For $n = 4$ that is $4 \cdot 5 / 2 = 10$, the 1 + 2 + 3 + 4 of the table. The expression $n^2/2 + n/2$ is $\Theta(n^2)$ by Theorem 2.1. The extra space is $O(n)$, for the result array `a`.

2.3.11 prefixAverage2: a running sum

The cure is to carry the sum forward: the sum of prefix `j` is the sum of prefix `j - 1` plus `x[j]`.

```
double[] prefixAverage2(double[] x) {
  int n = x.length;
  double[] a = new double[n];
  double total = 0;
  for (int j = 0; j < n; j++) {
    total += x[j];
    a[j] = total / (j + 1);
  }
  return a;
}
```

Two small changes decide everything: `double total = 0` has moved outside the loop, so the sum persists, and the inner loop has become the single line `total += x[j]`, which turns the sum of `x[0..j-1]` into the sum of `x[0..j]`.

`j`	`total += x[j]`	`total`	`a[j]`
start	—	0.0	—
0	+ 4	4.0	4.0 / 1 = 4.0
1	+ 8	12.0	12.0 / 2 = 6.0
2	+ 6	18.0	18.0 / 3 = 6.0
3	+ 2	20.0	20.0 / 4 = 5.0

Table 2.17: Trace of `prefixAverage2([4, 8, 6, 2])`.

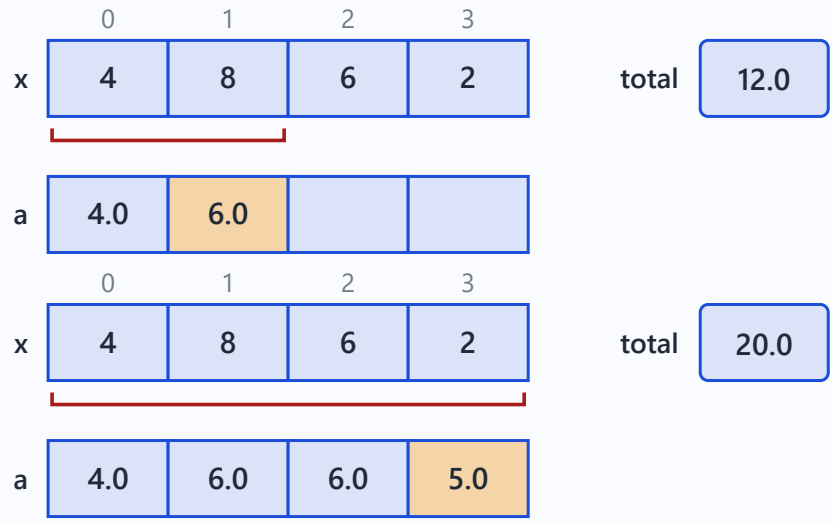


Figure 2.16: Two moments from the trace of `prefixAverage2`. Top: `j = 1`; only `x[1] = 8` is added to the previous 4.0, giving `total = 12.0` and `a[1] = 6.0`. Bottom: the last iteration, `total = 20.0`, `a[3] = 5.0`.

The result, `[4.0, 6.0, 6.0, 5.0]`, matches `prefixAverage1`, with 4 additions instead of 10. Each of the n iterations does one addition, one division and one assignment, so `prefixAverage2` runs in $\Theta(n)$ time, again with $O(n)$ extra space for the result.

Common mistake

Putting `double total = 0` inside the loop of `prefixAverage2`. Then `total` resets every iteration and the code computes `a[j] = x[j] / (j + 1)`: for `x = [4, 8, 6, 2]`, `[4.0, 4.0, 2.0, 0.5]`. The whole point of a running sum is that it survives between iterations.

2.3.12 Comparing the two solutions

Here are the addition counts side by side:

n	`prefixAverage1`: $n(n+1)/2$	`prefixAverage2`: n
10	55	10
100	5,050	100
1,000	500,500	1,000
1,000,000	500,000,500,000	1,000,000

Table 2.18: Additions performed for prefix averages.

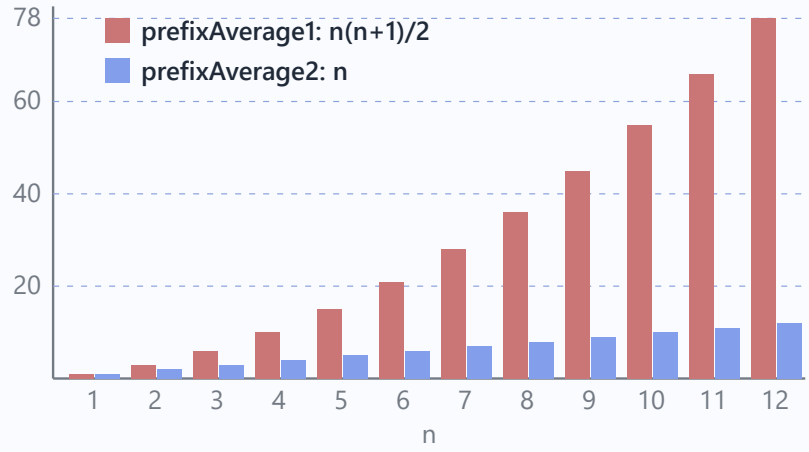


Figure 2.17: Additions for $n = 1 \dots 12$. The bars of `prefixAverage1` climb quickly as $n(n+1)/2$; those of `prefixAverage2` grow linearly as n.

At $n = 10$ the factor of 5.5 may seem minor. At $n = 10^6$, though, `prefixAverage1` performs about $5 \cdot 10^{11}$ additions against 10^6; at 10^9 additions per second, that is about 500 seconds (over 8 minutes) versus 1 millisecond. The ratio, $(n+1)/2$, keeps growing with n.

The gap comes from removing repeated work: `prefixAverage2` remembers the previous sum, `prefixAverage1` forgets it and starts over. Saving a result and reusing it underlies many efficient algorithms, and the first step in speeding one up is often to ask: "Am I doing the same work more than once?"

Both use the same space — an n-cell result array and a few variables — so the speedup cost no memory.

2.4 Space, correctness and robustness

So far we have measured only time. Three criteria of a good algorithm remain (2.1.2): memory, correctness and robustness. Memory is measured with the same tools as time; correctness and robustness are demonstrated and tested.

2.4.1 Space complexity

Space complexity describes how an algorithm's memory use grows with the input size, in Big O, with constants and smaller terms dropped.

The input itself is usually not counted. The array passed to `arrayMax` was in memory before the call; the method's own memory is just `currentMax` and `i`. Memory used beyond the input is **extra space**, and this is what analyses usually measure — though they should say whether the input is counted.



Figure 2.18: Three space classes (left to right). *O(1)*: a single variable. *O(n)*: an array of n cells. *O(n^2)*: an $n \times n$ matrix.

Figure 2.18 shows the three basic classes:

- ***O(1)*, constant space:** a few variables, independent of the input size. `arrayMax`, `sum`, `linearSearch`, `binarySearch` and `hasDuplicates` use only such variables (the maximum, the sum, indices, `lo`, `hi`, `mid`).
- ***O(n)*, linear space:** a new array the size of the input, like the result array `a` of `prefixAverage1` and `prefixAverage2`.
- ***O(n^2)*, quadratic space:** an $n \times n$ matrix, such as a table of distances between n cities. For $n = 10{,}000$ that is a hundred million cells — about 400 MB of 4-byte `int`s.

Memory efficiency matters most on constrained systems — embedded devices, phones, servers handling thousands of requests at once — but no memory is unlimited, and a quadratic-space algorithm exhausts even the largest machine once n is big enough.

An easily overlooked item is the **call stack**. Each call pushes a frame (2.3.4) that stays until the method returns. A loop uses one frame, but a method that calls itself — a recursive method — has as many frames open as its call depth. A recursion n levels deep uses $O(n)$ extra space without allocating a single array.

Example 2.8

Two ways to reverse an array:

1. **With a new array.** Allocate `b = new int[n]` and write `b[i] = a[n - 1 - i]`. Time $O(n)$, extra space $O(n)$.
2. **In place.** Start `i` at the front and `j` at the back; swap `a[i]` and `a[j]`, increment `i`, decrement `j`, and continue while `i < j`. The swap needs one temporary variable. Time $O(n)$, extra space $O(1)$.

Same time, different memory. The in-place version, however, modifies its input; if the original is needed later, a new array is unavoidable.

Common mistake

Forgetting the returned array, or counting the input, when computing extra space. `prefixAverage2` seems to need only a few variables, but the `double[n]` result array is new, so its extra space is $O(n)$. Conversely, the array `arrayMax` receives is not part of its extra space, which is $O(1)$.

2.4.2 Correctness

A fast algorithm with wrong results is worthless, so correctness comes before cost.

Definition 2.6

An algorithm is **correct** if, for every valid input, it produces the expected output in finite time. It must *halt*, and when it halts its result must be *right*.

"Every valid input" is the crux: inputs are infinite and cannot all be tried. Four tools help:

- **Test cases.** Run the algorithm on inputs with known answers — `arrayMax({3, 8, 2, 9, 4})` must return 9. Tests are excellent at finding bugs but prove nothing: an untested input may still fail.
- **Edge cases.** Bugs hide at the boundaries. Always try an empty input, a single element, extreme values, identical elements, and sorted or reverse-sorted arrays. For `arrayMax`: a one-element array, the maximum at the front, back and middle, and all-negative numbers.
- **Loop invariant.** A property that holds on every iteration, from which correctness is proved for all inputs, unlike testing (2.4.3).
- **Counterexample.** A single input proves an algorithm wrong. Correctness is a claim about all inputs; incorrectness needs just one.

Example 2.9

Is this version of `arrayMax` correct?

```
int currentMax = 0;
for (int i = 0; i < a.length; i++) if (a[i] > currentMax) currentMax = a[i];
return currentMax;
```

1. **Testing.** It returns 9 for `{3, 8, 2, 9, 4}` and 3 for `{1, 2, 3}`; every test with positive numbers passes.
2. **Suspicion.** The initial 0 is not an element of the array. If all elements are negative, none can beat `currentMax`.
3. **Counterexample.** `{-3, -1, -7}` returns 0. The answer is −1, and 0 is not even in the array.

One counterexample settles it. The fix is to start from an element of the array, `currentMax = a[0]` — exactly how `arrayMax` is written in 2.1.6.

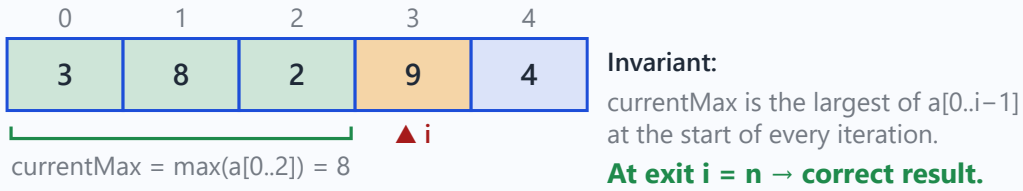


Figure 2.19: The loop invariant for `arrayMax`. At the start of the iteration with `i = 3`, the processed part is `a[0..2]` and `currentMax = 8` is its largest element. When the loop ends, `i = n` and the processed part is the whole array.

2.4.3 Proof by loop invariant

A test says “it works on these inputs”; a proof says “it works on every input.” For loops, the handiest proof tool is the loop invariant.

Definition 2.7

A **loop invariant** is a statement that is true at the start of every iteration of a loop. A proof by invariant has three steps:

- **Initialization:** the statement is true before the first iteration.
- **Maintenance:** if it is true at the start of an iteration, it is still true when the iteration ends (at the start of the next).
- **Termination:** when the loop ends, the statement shows that the algorithm is correct.

This is induction in disguise: initialization is the base case, maintenance the inductive step, and termination links the statement to the result.

Theorem 2.3 (Correctness of arrayMax)

For every array `a` with at least one element, `arrayMax(a)` returns the largest element of the array.

Proof. The invariant: *at the start of each iteration,* `currentMax` *is the largest of* `a[0..i-1]`.

- **Initialization.** Before the first iteration, `i = 1` and `currentMax = a[0]`, the largest (and only) element of `a[0..0]`.
- **Maintenance.** Suppose `currentMax` is the largest of `a[0..i-1]`. If `a[i]` is larger, `currentMax = a[i]`, which beats the old maximum and so all earlier elements: it is the largest of `a[0..i]`. Otherwise `currentMax` is unchanged and, being at least `a[i]`, remains the largest of `a[0..i]`. After `i++`, this reads “the largest of `a[0..i-1]`.”
- **Termination.** `i` grows by 1 per iteration and the loop stops at `i = n`, so it ends in finite time, and the invariant then says `currentMax` is the largest of `a[0..n-1]` — the whole array. The method returns it. ■

Figure 2.19 shows the invariant for `{3, 8, 2, 9, 4}` at `i = 3`: `currentMax = 8` is the maximum of the green part, `a[0..2] = {3, 8, 2}`; the next element, 9, is larger, so the iteration ends with `currentMax = 9`, the maximum of `a[0..3]`.

The theorem requires “at least one element”: an empty array has no `a[0]`, and initialization fails. This is not a gap in the proof but a precondition of the method; what should happen when it is violated is a question of robustness.

Common mistake

Thinking an invariant must hold mid-iteration. It must be true at the *start* of each iteration and may be briefly broken inside one. In `prefixAverage2`, "`total` is the sum of `x[0..j-1]`" breaks after `total += x[j]` and is restored by `j++` — which is just what the maintenance step shows.

2.4.4 Robustness

Correctness concerns valid inputs, but real programs also meet invalid ones: an empty file, a missing line, a malformed number. **Robustness** is the ability to cope with them. A robust algorithm neither crashes nor silently produces a wrong result outside the normal range; it either handles the input sensibly or reports the problem clearly.

Robustness is tested from four angles:

Situation	What is tested	Example
Invalid inputs	Handling inappropriate or unexpected inputs	`null`, negative size
Faulty data	Handling corrupt or incomplete data	missing line, wrong format
Large data sets	Processing very large inputs	10^8 records
Overload	Behavior under heavy load	thousands of simultaneous requests

Table 2.19: Situations in which robustness is tested.

Each is a real source of failures. Given an empty array, `arrayMax` throws `ArrayIndexOutOfBoundsException` at `a[0]`; given `null`, `NullPointerException`. Either stops the program, but neither message states the cause. The binary search overflow (2.3.7) is a large-data robustness problem: flawless on small arrays, broken beyond a billion elements.

A robust version checks its precondition and reports violations clearly:

```
int arrayMaxChecked(int[] a) {
  if (a == null || a.length == 0)
    throw new IllegalArgumentException("Array must not be empty");
  int currentMax = a[0];
  for (int i = 1; i < a.length; i++)
    if (a[i] > currentMax)
      currentMax = a[i];
  return currentMax;
}
```

The first line catches both invalid cases. `a == null` comes first, and `||` skips its right side when the left is true, so the `length` of a `null` array is never read. Invalid input raises `IllegalArgumentException` with a message saying what is wrong; the rest is identical to `arrayMax`.

The check costs at most two comparisons, $O(1)$, and the method stays $\Theta(n)$. Robustness is often this cheap, so checking preconditions is a good habit, especially in methods others will call.

What to do with an empty array is a design decision: throw an exception, or return a special value — Java's `OptionalInt` exists for this. The one wrong choice is a silent, meaningless value, like the 0 that the faulty version in Example 2.9 returns, leading the caller to believe the array's largest element is 0.

Common mistake

Equating robustness with "not crashing." A program that swallows an error and carries on with a wrong value doesn't crash, but it isn't robust; a robust one notices the problem and reports it. A silent wrong result is more dangerous than a loud crash, because it slips unnoticed into other computations.

2.5 Common mistakes

Table 2.20 collects the chapter's most important mistakes.

Misconception or mistake	The truth
"O = worst case"	O is an upper bound; best and worst case are separate ideas, and O, Ω, Θ apply to any case.
"$O(1)$ is always fast"	The constant can be large: an $O(1)$ operation of 1000 steps per call is slower than an n-step $O(n)$ one while $n < 1000$.
"Nested loops = n^2"	With an inner `j *= 2` it is $n \log n$; with a fixed-length inner loop, $O(n)$.
"The base of the logarithm matters"	$\log_2 n$ and $\log_{10} n$ differ by a constant factor; both are $O(\log n)$.
`(lo + hi) / 2` **overflow**	The `int` overflows on very large arrays; `lo + (hi - lo) / 2` is safe.
Comparing arrays with `a == b`	`==` compares references; use `Arrays.equals(a, b)` for contents.
Ignoring warm-up and noise when timing	The first run is slow because of the JIT; repeat and take the median.
Measuring the worst case with the wrong input	An array of zeros is the best case of `hasDuplicates`; the worst case needs distinct values.

Table 2.20: Common misconceptions and mistakes in algorithm analysis.

The array comparison mistake is common when testing methods like `hasDuplicates`. After `int[] a = {1, 2}; int[] b = {1, 2};`, `a == b` is `false`: the variables refer to two separate arrays, and `==` asks "same array?", not "same contents?". `Arrays.equals(a, b)` compares element by element and returns `true`, in $O(n)$ time; `==` compares two references in $O(1)$.

The lesson of "$O(1)$ is always fast" applies to every class. A linear algorithm of $1000n$ steps beats a quadratic one of n^2 steps only when $n > 1000$; below that the quadratic one is faster. Asymptotic analysis says which wins as inputs grow; for small inputs, the constants matter too.

2.6 Summary

- An **algorithm** is a method of finite, precise and effective steps that is correct for every valid input, judged by time, space, correctness and robustness.
- **Empirical analysis** times code with `System.nanoTime()`, taking the median of repeated runs after a warm-up. In a **doubling experiment**, $T(2n)/T(n) \approx 2^k$ means $T(n) \approx c \cdot n^k$. It depends on hardware, covers only the inputs tried, and needs complete code.
- **Counting primitive operations** is hardware-independent and valid for all inputs. `arrayMax` performs $3n + 1$ operations at best and $4n$ at worst, both $\Theta(n)$.
- $f(n) = O(g(n))$: for some $c > 0$ and n_0, $f(n) \leq c \cdot g(n)$ for every $n \geq n_0$ (upper bound); Ω is a lower bound, Θ both (tight bound). Constant factors and smaller terms are dropped, a degree-d polynomial is $\Theta(n^d)$, and the base of a logarithm doesn't matter.
- **Best, average and worst case** specify which input's cost we study, independently of the bound notations. The worst case is a guarantee, the average case assumes a distribution, the best case alone misleads. **Amortized analysis** divides the total cost of a long sequence by its length; adding to a growable array is amortized $O(1)$.
- Common classes, slowest-growing first: $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(2^n)$. At 10^9 operations per second, 2^n steps for $n = 100$ take about $4 \cdot 10^{13}$ years.
- **Loop patterns**: single loop $O(n)$; doubling or halving loop $O(\log n)$; two full nested loops $O(n^2)$; n outer around $\log n$ inner $O(n \log n)$. Consecutive blocks add; nested loops multiply.
- **Linear search** is $O(n)$ and needs no sorting. **Binary search** halves the range of a sorted array each iteration, making at most $\lfloor \log_2 n \rfloor + 1$ iterations: $O(\log n)$. Compute the middle as `lo + (hi - lo) / 2`.
- `hasDuplicates` compares all pairs: $n(n - 1)/2$ comparisons in the worst case, $O(n^2)$. For **prefix averages**, `prefixAverage1` re-sums each prefix with $n(n + 1)/2$ additions, $O(n^2)$; `prefixAverage2` carries the sum with n additions, $O(n)$.
- **Space complexity** usually measures extra memory: a few variables $O(1)$, a new array $O(n)$, an $n \times n$ matrix $O(n^2)$. Recursion also counts the call stack.
- **Correctness** is shown with tests, edge cases and loop invariants (initialization, maintenance, termination); one counterexample proves an algorithm wrong. A **robust** algorithm detects invalid input and reports it clearly.

Topic	Key points
Empirical analysis	time depends on hardware and input; doubling experiment: $T(2n)/T(n) \approx 2^k \rightarrow T(n) \approx c \cdot n^k$
Counting primitive operations	assignment, comparison, arithmetic, array access; hardware-independent, valid for all inputs
O, Ω, Θ	upper, lower and tight bounds; constant factors and smaller terms are dropped
Cases	best, average, worst; worst case for guarantees, amortized for long sequences of operations

Topic	Key points
Loop patterns	single loop $O(n)$ · nested $O(n^2)$ · halving $O(\log n)$ · outer n, inner $\log n \to O(n \log n)$
Choosing an algorithm	prefix averages $O(n^2) \to O(n)$; binary search on a sorted array $O(\log n)$
Space, correctness	extra memory is measured in Big O too; a fast algorithm that is not correct and robust is worthless

Table 2.21: Chapter summary.

These tools will be used to compare the operations of every data structure that follows. Table 2.22 previews the costs of common operations on structures covered later in the course, to show how differently the same operation can cost.

Data structure	Access *i*th element	Search	Insert	Delete
Array	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Linked list	$O(n)$	$O(n)$	$O(1)$†	$O(1)$†
Stack	top only	$O(n)$	$O(1)$ push	$O(1)$ pop
Queue	front only	$O(n)$	$O(1)$ enqueue	$O(1)$ dequeue
Map / set (hash)	—	$O(1)$*	$O(1)$*	$O(1)$*

Table 2.22: Operation costs on the fundamental data structures. † If the position (the previous node) is known. * Average case; $O(n)$ in the worst case.

The first row ties back to this chapter. Accessing the ith element is an address computation, as in `first` : $O(1)$. Searching an unsorted array is linear search, $O(n)$; a sorted array allows binary search in $O(\log n)$. Inserting or deleting in the middle shifts the elements that follow: $O(n)$.

2.7 Key terms

Each term links to the course glossary, which gives its Turkish equivalent and its use in other chapters.

algorithm · algorithm analysis · complexity · time complexity · space complexity · empirical analysis · doubling experiment · primitive operation · asymptotic · Big O · Big Omega · Big Theta · best case · average case · worst case · amortized analysis · loop · linear search · binary search · prefix averages · stack frame · reference · correctness · invariant · robustness

2.8 Test Yourself

1. How many times is `currentMax` updated in `arrayMax({5, 1, 7, 7, 2})`? With the counting rules of Table 2.3, how many primitive operations are performed?

   Answer

   1 > 5 false, 7 > 5 true (update), 7 > 7 false, 2 > 7 false: a single update, $u = 1$, since the second 7 is only equal. Total $3n + 1 + u = 3 \cdot 5 + 1 + 1 = 17$.

2. For $f(n) = 4n + 7$, give c and n_0 showing $f(n) = O(n)$. Is $f(n) = \Theta(n)$?

   Answer

   With $c = 5$, $4n + 7 \le 5n$ for $n \ge 7$, so $n_0 = 7$ ($c = 11$, $n_0 = 1$ also works). Since $4n + 7 \ge 4n$ for every $n \ge 1$, $f(n) = \Omega(n)$, and together $f(n) = \Theta(n)$.

3. Which are true? (a) $n^2 + 100n = O(n^2)$ (b) $2^{n+1} = O(2^n)$ (c) $n \log n = O(n)$ (d) $4^n = O(2^n)$ (e) $\log_{10} n = \Theta(\log_2 n)$

   Answer

   (a), (b) and (e). (a): $n^2 + 100n \le 101n^2$ for $n \ge 1$. (b): $2^{n+1} = 2 \cdot 2^n$, a constant factor. (e): the logarithms differ by a constant factor. (c) is false: $n \log n / n = \log n$ grows without bound. (d) is false: $4^n / 2^n = 2^n$ grows without bound.

4. How many iterations does `for (int i = 1; i <= n; i *= 3) s++;` make for $n = 81$? What is its complexity?

   Answer

   `i` is 1, 3, 9, 27, 81 (the condition is `<=`, so 81 counts), then 243 ends the loop: 5 iterations, $\log_3 81 + 1$. $O(\log n)$; base 3 does not change the class.

5. What is `s` after this code with $n = 10$? What is the complexity?

```
for (i = 0; i < n; i++) for (j = 0; j < 5; j++) s++;
for (k = 0; k < n; k++) for (m = 0; m < k; m++) s++;
```

   Answer

   First block: 10 outer times a fixed 5 inner, 50; $O(n)$. Second: $0 + 1 + \ldots + 9 = 10 \cdot 9 / 2 = 45$; $O(n^2)$. So `s = 95`, and the larger block dominates: $O(n^2)$.

6. For `a = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91}`, how many iterations do `binarySearch(a, 91)` and `binarySearch(a, 2)` make? Give `lo`, `hi` and `mid` for each.

   Answer

   For 91: (0, 9, `mid` 4, 16 < 91) → `lo = 5`; (5, 9, 7, 56) → `lo = 8`; (8, 9, 8, 72) → `lo = 9`; (9, 9, 9, 91) found, returns 9. 4 iterations. For 2: (0, 9, 4, 16 > 2) → `hi = 3`; (0, 3, 1, 5 > 2) → `hi = 0`; (0, 0, 0, 2) found, returns 0. 3 iterations. Both are within $\lfloor \log_2 10 \rfloor + 1 = 4$.

7. Linear search runs on 9 elements. If the target is always present and equally likely at each position, how many comparisons are made on average? How many if it is absent?

   Answer

   A target at position k (from 1) costs k comparisons, so the average is $(1 + 2 + \ldots + 9)/9 = 45/9 = 5$, matching $(n + 1)/2$. An absent target costs all 9.

8. A doubling experiment gives 40, 81, 160 and 322 ms for n = 1000, 2000, 4000 and 8000. What is the growth class? About how long will $n = 16{,}000$ take?

   Answer

   The ratios $81/40 \approx 2.03$, $160/81 \approx 1.98$, $322/160 \approx 2.01$ give $2^k \approx 2$, so $k = 1$: linear. $n = 16{,}000$ should take about $2 \cdot 322 = 644$ ms.

9. A student starts `arrayMax` with `currentMax = 0` and loops from `i = 0`. Give a counterexample. Which invariant fails at initialization?

   Answer

   `{-4, -2, -9}` returns 0 instead of −2. The invariant "at the start of each iteration, `currentMax` is the largest of `a[0..i-1]`" fails: initially `i = 0`, `a[0..-1]` is empty and has no largest element, and 0 is not in the array. With initialization broken, the proof collapses and arrays of negative numbers give wrong results.

10. What is the extra space of each? (Input not counted.) (a) `sum` (b) `prefixAverage1` (c) a method that builds, from an n-element array, the $n \times n$ matrix of `a[i] + a[j]` for every pair `(i, j)` (d) `countSteps`

    Answer

    (a) $O(1)$: just `s` and `x`. (b) $O(n)$: the result array `a`. (c) $O(n^2)$: n^2 cells. (d) $O(1)$: just `k` and `i`.

2.9 Further reading

- M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, *Data Structures and Algorithms in Java* (6th ed., 2014), Chapter 4: empirical analysis, counting primitive operations, Big O, Omega and Theta, and the prefix averages example, in an order close to these notes and with Java code.
- R. Sedgewick and K. Wayne, *Algorithms* (4th ed., 2011), Section 1.4: the doubling experiment and predicting running time from observations, with detailed examples of empirical and mathematical analysis working together.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms* (4th ed., 2022), Chapters 2 and 3: correctness proofs by loop invariant and the formal definitions of asymptotic notation, in more mathematical language.
- J. Bloch, "Extra, Extra — Read All About It: Nearly All Binary Searches and Mergesorts are Broken" (Google Research Blog, 2006): a short account of how the middle-index overflow surfaced in real software.

2.10 References

1. J. Bloch, "Extra, Extra — Read All About It: Nearly All Binary Searches and Mergesorts are Broken", Google Research Blog, June 2, 2006.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022.
3. M. T. Goodrich, R. Tamassia, M. H. Goldwasser, *Data Structures and Algorithms in Java*, 6th ed., Wiley, 2014.
4. R. Sedgewick, K. Wayne, *Algorithms*, 4th ed., Addison-Wesley, 2011.

PART I · CHAPTER 3

Arrays

The array is the first and most basic data structure a programmer meets. It keeps elements of one type side by side in memory and reaches any of them by number in one step. Most other structures are either built on arrays or invented to fix something arrays do badly, so it pays to know exactly how an array sits in memory and which operations it makes fast or slow. We start with the array's memory layout, then Java arrays: declaring and copying them, the stack and the heap, and the usual mistakes. Array algorithms follow — mean, median, mode, reversal, the Caesar cipher — with code traces, then multidimensional arrays, matrix operations, and a small Tic Tac Toe game. The last part works out the time and space cost of search, insertion, deletion, and two simple sorts.

Code for this chapter: `github.com/sercankulcu/data-structures-java/Bolum03/src`

LEARNING OUTCOMES

The array model
Explain how arrays are laid out in memory and indexed, and why an element's address takes one step to compute.

Arrays in Java
Declare and use one- and multidimensional arrays correctly, minding the difference between reference and copy, and between shallow and deep copy.

Algorithms
Implement search, insertion, deletion, reversal, and sorting on arrays, and trace them by hand.

Cost analysis
State and compare the time and space costs of array operations, with reasons.

3.1 The idea of an array

Picture a program that stores the exam scores of 40 students. Forty separate variables (`score1`, `score2`, ..., `score40`) are tedious and nearly useless: averaging them means writing out forty names, and a 41st student means changing the program. We want all forty scores under one name, each reachable by its position. That is what an array provides.

3.1.1 What is an array?

An **array** stores many values of the same type, in order, under one name. Each value is an **element**, and its position number is its **index**. In Java, C, and many other languages indices start at 0: the first element has index 0, the second index 1.

> **Definition 3.1**
>
> An **array** is a data structure that holds a fixed number of elements of the same type in **contiguous** memory slots, one right after another. The elements of an array of n elements are accessed by the indices 0, 1, ..., $n - 1$. The number n is the array's **size**, or length.

Each part of the definition matters:

- **Same type.** An `int` array holds only integers, a `String` array only strings. So every element takes the same number of bytes, which makes an element's location easy to compute.
- **Fixed size.** The size is set at creation and never changes. A five-element array cannot grow a sixth cell; for more room we create a bigger array and copy the elements into it.
- **Contiguous memory.** Elements are packed back to back with no gaps: element 1 directly follows element 0, element 2 follows element 1.

Fixed size does not mean fixed contents: the 4 in cell 2 can become 40 at any time. What is fixed is the number of cells, so insertion and deletion mean something special for arrays. Inserting in the middle shifts the elements to its right one cell over to make room; deleting shifts them left to close the gap. Sections 3.5.5 and 3.5.6 work out what these shifts cost.



Figure 3.1: A five-element array: indices (0 to 4) on top, elements (13, −1, 4, 2, 7) below. The length is 5; the first index is 0 and the last is 4.

Figure 3.1 shows the most useful picture of an array: a row of numbered boxes, values inside, indices above. Don't confuse the two. Here the element at index 2 is 4, while the element whose value is 2 sits at index 3.

Arrays appear wherever many values of one type are processed together:

- **Matrices.** Tables of numbers in rows and columns live in two-dimensional arrays (3.4).
- **Images.** A digital image is a grid of pixels, each with a color value, stored in a two-dimensional array.
- **Sound.** A recording is a sequence of amplitude samples, taken thousands of times a second and stored in time order in a one-dimensional array.

3.1.2 Indexing and address arithmetic

To reach an element, write the array's name and the index in square brackets: `arrayName[index]`. This code creates a five-element array and reads its first and last elements:

```
int[] arr = {7, 2, 9, 5, 1};
int first = arr[0];      // 7
int last = arr[arr.length - 1];      // 1
```

`arr[0]` is 7. The last index is one less than the length, which every Java array carries in its `length` field; here `arr.length` is 5, so the last element is `arr[arr.length - 1]`, that is, `arr[4]` = 1. Going through `length` instead of writing 4 keeps the code correct if the size ever changes.

Indices start at 0 because of how elements are found in memory. Memory is a sequence of bytes, each with a numbered address. Creating an array reserves a contiguous region big enough for all its elements; the address of its first byte is the array's **base address**. An `int` takes 4 bytes, so element 0 is at the base address, element 1 four bytes later, element 2 eight bytes later. In general, element *i* is at

> *address(i) = base + i × element size*

Starting at 0, the first element's address is the base itself and the formula needs no "−1." An index is really the element's distance from the start, counted in elements.

Example 3.1

Suppose an `int` array starts at address 1000. With 4 bytes per `int`, the elements are at:

1. `arr[0]` : 1000 + 0 × 4 = 1000.
2. `arr[1]` : 1000 + 1 × 4 = 1004.
3. `arr[2]` : 1000 + 2 × 4 = 1008.
4. `arr[4]` : 1000 + 4 × 4 = 1016.

Any index takes one multiplication and one addition; reaching element 2 never visits elements 0 and 1.

This is the array's greatest strength. The time to reach an element depends neither on the array's size nor on the element's position: `arr[i]` takes as long in an array of five elements as in one of five million. This is **random access** — "random" meaning any element you like, directly — and it costs *O(1)* (3.5.2).

Common mistake

Reaching for the last element with `arr[arr.length]`. A five-element array has indices 0 through 4 and no cell 5. Since people count from 1, the "fifth element" seems to have index 5, but its index is 4. Java catches the error at run time and throws an exception (3.2.12).

3.1.3 Kinds of arrays

Arrays are named by how many directions their elements extend in; Figure 3.2 shows three kinds.

- **One-dimensional array.** A single row of elements, each reached with one index, like every example so far: `int[] a = new int[5];`
- **Two-dimensional array.** Rows and columns, like a table, also called a **matrix**. Each element takes two indices, row first, then column: `int[][] m = new int[3][3];`
- **Multidimensional array.** Anything needing more than two indices, such as video frames indexed by time × row × column: `int[][][] k = new int[3][3][3];`

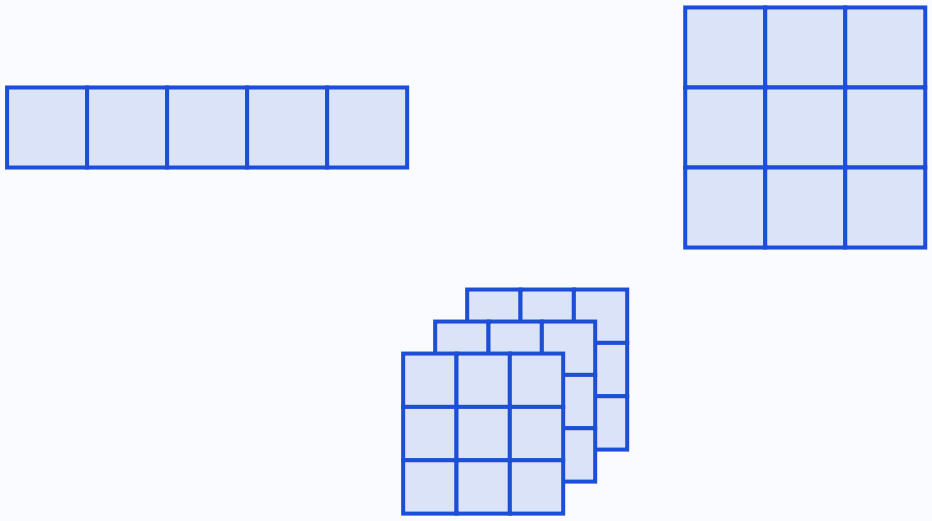


Figure 3.2: Kinds of arrays. Left, one-dimensional: a single row. Middle, two-dimensional: 3 rows and 3 columns. Right, three-dimensional: three stacked 3 × 3 layers.

Each dimension adds a pair of brackets to the declaration, and the element count is the product of the dimensions: `new int[3][3]` holds 9, `new int[3][3][3]` holds 27. How Java actually stores multidimensional arrays may surprise you; see Section 3.4.2.

3.1.4 Strengths and weaknesses

Choosing a structure means knowing where it is strong and where it is weak. Table 3.1 does this for arrays.

Strengths	Weaknesses
Fast, direct access by index: *O(1)*.	Fixed size; a full array cannot grow, so a new one is created and the elements copied.
Groups the data and keeps insertion order.	Inserting or deleting in the middle requires shifting elements: *O(n)*.
Holds primitives (`int` , `char` …) as well as objects.	Unused capacity is wasted.
Contiguous memory: no extra references, cache-friendly.	No built-in methods for sorting or searching; `java.util.Arrays` supplies them.
—	Java arrays are covariant, which brings both flexibility and a hole in type safety.

Table 3.1: Strengths and weaknesses of arrays.

Most strengths come from contiguous memory. Access is fast because of the address arithmetic of Section 3.1.2. “No extra references” is clearest next to a linked list, where every element carries a reference to the next one, costing memory. In an array the next element is simply next door, so all the memory goes to data.

Contiguity also makes arrays **cache**-friendly. The processor fetches memory in blocks and keeps them in a small, very fast cache beside it. Reading element 0 brings elements 1, 2, and 3 from the same block along, so a pass through the array finds most elements already waiting. Of two algorithms taking the same number of steps, the one working on an array is therefore usually faster in practice.

Most weaknesses share that source. With no gaps between elements, inserting in the middle requires making room. A full array cannot gain a cell at the end, because the memory right after it may belong to other data. And if the element count is unknown in advance, we must allocate generously and waste the unused cells.

The last row of the table, **covariance**, needs a closer look.

3.1.5 Array covariance and ArrayStoreException

Java classes are related by subtyping: `String` is a subtype of `Object` , so every `String` is an `Object` and `Object o = "Ann";` is legal. Is an array of `String` s then a subtype of an array of `Object` s? Java says yes.

Definition 3.2

Arrays are **covariant** if, whenever `S` is a subtype of `T`, `S[]` is a subtype of `T[]`. This property is called **array covariance**. In Java, arrays of reference types are covariant: `String[]` is a subtype of `Object[]`.

Covariance buys flexibility: one method `static void print(Object[] a)` accepts a `String[]`, an `Integer[]`, or any other array of references. But it has a price:

```
String[] names = {"Ann", "Alice"};
Object[] objects = names; // compiles
objects[0] = 42;   // compiles, but
                   // throws at run time!
```

Line 1 creates a two-element `String` array. Line 2 gives `objects` a reference to the same array — legal thanks to covariance — so two variables now point to one array on the heap. On line 3 the compiler sees an `Object[]` receiving 42 (boxed as an `Integer`, an `Object`) and has no complaint. But the object really is a `String[]` and cannot hold an integer. The Java Virtual Machine (JVM) notices at the moment of the assignment and throws `ArrayStoreException: java.lang.Integer`.

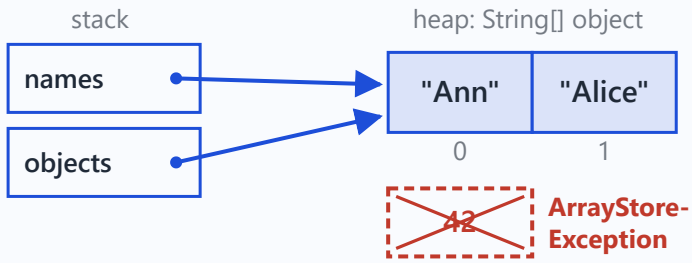


Figure 3.3: `names` and `objects` live on the stack and point to the same `String[]` object on the heap. The assignment `objects[0] = 42` is rejected at run time with an `ArrayStoreException`.

An error caught at compile time shows up on the programmer's desk; one caught at run time may surface months later in front of a user. Array covariance moves a type error from the first to the second — that is the hole in type safety.

Java's later **generics** were designed to close it. Generic lists are not covariant: `List<String>` is not a subtype of `List<Object>`, so `List<Object> l = new ArrayList<String>();` does not compile, and the same mistake is caught by the compiler.

Common mistake

Assuming that code that compiles has the right types. With arrays it may not: an `Object[]` variable can point to an array of a narrower type, and a store of the wrong type is detected only at run time. Any method that takes an `Object[]` and writes into it carries this risk.

3.2 Arrays in Java

Every language has arrays, but each declares them and lays them out in its own way. In Java an array is an object, and that one fact decides where it lives, what an assignment copies, and what a method receives when passed an array. This section goes from syntax down to memory layout.

3.2.1 Declaring arrays and default values

A Java array is declared and created with this pattern:

```
DataType[] arrayName = new DataType[arraySize];
```

- `DataType` is the element type, such as `int`, `double`, or `String`; the `[]` says the variable refers to an array, not a single value.
- `arrayName` names the reference variable that points to the array, for example `nums`.
- `arraySize` is the number of elements. It need not be a constant; `5`, `n`, or `2 * n`, computed at run time, all work.

The left side declares a **reference** variable — a value that records where an object is in memory, not the object itself. The right side, `new DataType[arraySize]`, creates a new array object, and the assignment stores its reference in the variable. This distinction underlies the discussion of copying in Section 3.2.6.

```
int[] nums = new int[5];
// → [0, 0, 0, 0, 0]
```

No values have been assigned here, but the five elements are not empty: every element of an array created with `new` gets its type's **default value**, 0 for `int`. Table 3.2 lists the defaults.

Type	Default value
`int`, `long`, `short`, `byte`	`0`
`double`, `float`	`0.0`
`boolean`	`false`

Type	Default value
`char`	`'\u0000'` (the character with code 0)
object (`String` ...)	`null`

Table 3.2: Default element values of an array created with `new`.

Here arrays differ from local variables: reading an unassigned local `int x;` is a compile error, but array elements always have a value, because the JVM zeroes every cell at creation. That zeroing is work too, so creating an array of *n* elements takes *O(n)* time.

The `char` default, `'\u0000'`, is an invisible character, not a space. A `String` array defaults not to empty strings (`""`) but to `null`, the reference that points to no object; calling a method through such an element (`names[0].length()`) throws a `NullPointerException`.

> **Common mistake**
>
> Believing that `String[] names = new String[3];` holds three empty strings. It holds three `null` s and no strings at all. Assign each cell a string before calls such as `names[i].length()` .

3.2.2 Access, initialization, and length

When the values are known in advance, list them in braces instead of filling cells one by one; this is an **initializer** list. The code below shows initialization, reading, writing, and `length` :

```
int[] nums = {10, 20, 30, 40, 50};   // 5 initializers → size 5
int first  = nums[0];            // 10
int second = nums[1];            // 20
nums[4] = 55;                        // last element updated
int size   = nums.length;     // 5

nums = new int[] {1, 2, 3};          // later assignment needs new int[]
int[] bad = new int[3] {1, 2, 3};   // compile error: size + list
```

- **Line 1:** The five values set the size; no size is written. Java creates the array, stores the values in order, and assigns the reference to `nums` .
- **Lines 2 and 3:** Reading: `nums[0]` is 10, `nums[1]` is 20.
- **Line 4:** Writing uses the same syntax on the left of an assignment; the last cell (index 4) changes from 50 to 55.
- **Line 5:** `nums.length` is 5. `length` is a **field**, not a method, so no parentheses follow it.
- **Line 7:** A bare initializer list is allowed only in a declaration; elsewhere it needs `new int[]` in front.

- **Line 8:** A compile error: the list already fixes the size, so you cannot also give one. Write `new int[] {1, 2, 3}`.

> **Common mistake**
>
> Writing `length()` for an array and `length` for a string. An array's length is a field (`a.length`), a `String`'s a method (`s.length()`); either slip is a compile error.

3.2.3 Code trace: reading and writing elements

Here is what that code does in memory. The traced lines:

```
int[] nums = {10, 20, 30, 40, 50};
int first  = nums[0];
int second = nums[1];
nums[4] = 55;
int size   = nums.length;
nums = new int[] {1, 2, 3};
```

Line	Action	Variables	`nums`
1	a 5-element array is created on the heap	—	[10, 20, 30, 40, 50]
2	`nums[0]` is read	`first` = 10	[10, 20, 30, 40, 50]
3	`nums[1]` is read	`second` = 20	[10, 20, 30, 40, 50]
4	55 is written to `nums[4]`	—	[10, 20, 30, 40, **55**]
5	`length` is read	`size` = 5	[10, 20, 30, 40, 55]
6	a new 3-element array is created and `nums` points to it	`size` is still 5	[1, 2, 3]

Table 3.3: Trace of the read/write code. The last column shows the array that `nums` points to.

Line 1 creates two things: a five-element `int[]` object on the heap, and the variable `nums`, which holds only the object's reference — the arrow in Figure 3.4, left.

Line 2 reads with the address arithmetic of Section 3.1.2, base + 0 × 4, in one step whatever the size; line 3 does the same for index 1. On line 4 the same syntax stands left of an assignment, so it writes: the last cell becomes 55. Line 5 reads the length, fixed when the array was created.

The last line teaches the most. `new int[] {1, 2, 3}` creates a *new* array, and `nums` now points to it. The old array still holds [10, 20, 30, 40, 55], but no reference leads to it, so the program can never reach it again. Java's **garbage collector** (GC) finds such unreachable objects and reclaims their memory.

Note too that `size` is still 5. Line 5 copied the *value* of `nums.length` at that moment; `size` is not tied to the array and does not follow its changes.

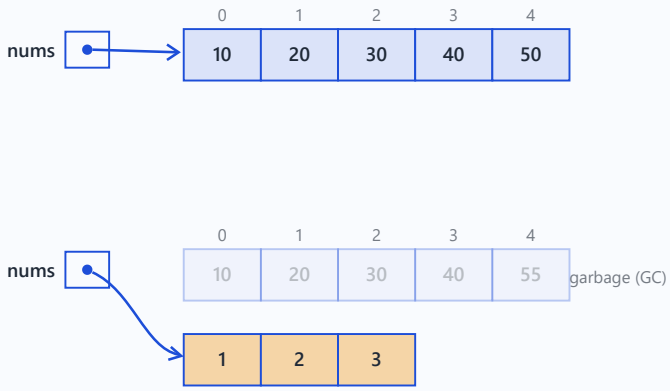


Figure 3.4: Two moments of the read/write trace. Left, after line 1: `nums` points to [10, 20, 30, 40, 50] on the heap. Right, after the last line: `nums` points to the new array [1, 2, 3]; the old array [10, 20, 30, 40, 55], drawn faded, is unreachable and may be collected.

3.2.4 Sample code: declare, fill, print

This complete program creates an array, fills it in a loop, and prints it two ways:

```
public class ArrayExample {
    public static void main(String[] args) {
        int[] nums = new int[5];            // 5-element array
        for (int i = 0; i < nums.length; i++)
            nums[i] = (i + 1) * 10;             // 10, 20, 30, 40, 50

        System.out.println("First element: " + nums[0]);
        System.out.println("Third element: " + nums[2]);

        System.out.print("Elements: ");
        for (int x : nums)                      // for-each: when no index is needed
            System.out.print(x + " ");
    }
}
```

The output:

```
First element: 10
Third element: 30
Elements: 10 20 30 40 50
```

This is the skeleton of most array code:

- **Create.** `new int[5]` gives five zeros.

- **Fill.** The loop runs `i` from 0 while `i < nums.length`, writing `(i + 1) * 10` into `nums[i]`: 10, 20, 30, 40, 50.
- **Print single elements.** `nums[0]` and `nums[2]`; the third element has index 2.
- **Print all elements.** The **for-each loop** `for (int x : nums)` reads "for each `x` in `nums`"; `x` takes each element's value in turn.

The loop's real gain over five assignments is independence from the size: it works unchanged for 500 elements, as long as the bound is `nums.length`. A bound of `i < 5` works today but, once the size changes, either fills half the array or runs off its end.

`System.out.println` ends its output with a line break; `System.out.print` does not, so the last loop prints all elements on one line, separated by spaces.

For-each reads better when no index is needed, but has two limits. It does not know the index, so "where did we find it?" needs a separate counter. And `x` is a copy of the value, so `x = 0` leaves the array unchanged. To write into an array, use an indexed `for`.

3.2.5 Code trace: fill and print

Table 3.4 traces the fill loop and the print statements.

Step	Condition / action	`nums`	Output
`new int[5]`	5 cells are allocated	[0, 0, 0, 0, 0]	—
`i = 0`	0 < 5 → `nums[0]` = 1 × 10	[10, 0, 0, 0, 0]	—
`i = 1`	1 < 5 → `nums[1]` = 2 × 10	[10, 20, 0, 0, 0]	—
`i = 2`	2 < 5 → `nums[2]` = 3 × 10	[10, 20, 30, 0, 0]	—
`i = 3`	3 < 5 → `nums[3]` = 4 × 10	[10, 20, 30, 40, 0]	—
`i = 4`	4 < 5 → `nums[4]` = 5 × 10	[10, 20, 30, 40, 50]	—
`i = 5`	5 < 5 false → loop ends	[10, 20, 30, 40, 50]	—
1st `println`	`nums[0]` is read	unchanged	First element: 10
2nd `println`	`nums[2]` is read	unchanged	Third element: 30
`print`	no line break	unchanged	Elements:
for-each	`x` = 10, 20, 30, 40, 50	unchanged	10 20 30 40 50

Table 3.4: Trace of `ArrayExample`.

The final test is easy to overlook: at `i` = 5, `5 < 5` is false, so `nums[5]` is never touched. With `<=`, the last iteration would write to a nonexistent cell 5 and the program would stop with an exception.

The body ran 5 times, one multiplication and one assignment each; with *n* elements, *n* times. Any loop that visits every element once costs *O(n)* (3.5).

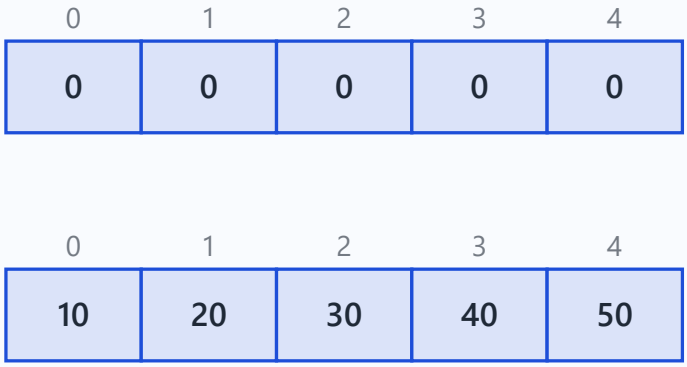


Figure 3.5: Start and end of the fill loop. Left, after `new int[5]` every cell is 0. Right, when the loop ends the array holds 10, 20, 30, 40, 50.

3.2.6 Copying arrays: reference or copy?

A Java array is an **object**: it lives on the heap, and a variable holds its reference, not the object. The most surprising consequence is that `=` does not copy an array:

```
int[] arr = {1, 2, 3, 4, 5};
int[] ref = arr;   // same object
int[] copy =
    Arrays.copyOf(arr, arr.length);
ref[0] = 99;
```

We trace it in four steps, numbering the heap arrays #1 and #2.

Example 3.2

1. `int[] arr = {1, 2, 3, 4, 5};` Array #1 is created on the heap; `arr`, on the stack, holds only its address.
2. `int[] ref = arr;` `=` copies the address, so `ref` points to #1 too. No new array exists; one array has two names.
3. `int[] copy = Arrays.copyOf(arr, arr.length);` `Arrays.copyOf` creates array #2 and copies #1's elements into it one by one, in $O(n)$. `copy` points to #2.
4. `ref[0] = 99;` `ref` points to #1, so #1's first cell becomes 99, and so does `arr[0]`. `copy[0]`, in a separate object, stays 1.

Result: `arr` = [99, 2, 3, 4, 5], `copy` = [1, 2, 3, 4, 5].

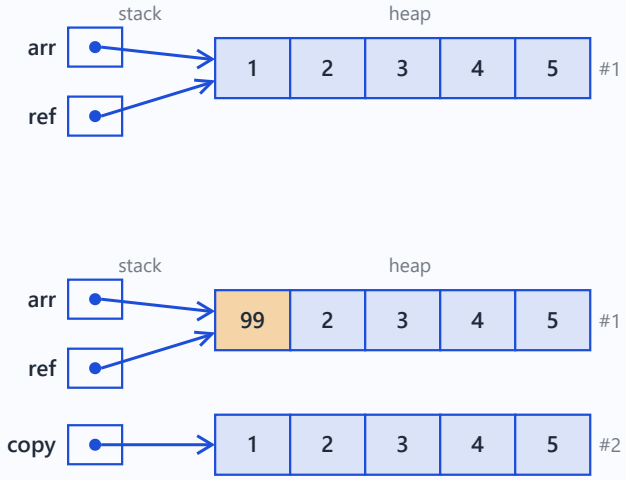


Figure 3.6: Reference versus copy. Left, after `ref = arr` both variables point to array #1 on the heap. Right, `Arrays.copyOf` has created a separate array #2; `ref[0] = 99` changes only #1, and the change is visible through `arr`.

Three common ways give a real copy, each creating a new array and copying element by element in *O(n)*:

- `Arrays.copyOf(arr, newSize)` returns an array of the requested size, padding with default values or truncating. This is also how a full array is grown.
- `arr.clone()` returns a same-size copy.
- `System.arraycopy(src, srcPos, dest, destPos, length)` copies a given number of elements into an existing array, starting at given positions.

The same rule holds for method calls: a method receives the array's reference, so a change it makes inside the array is visible to the caller. That is why `median` in Section 3.3.4 copies the array before sorting.

> **Common mistake**
>
> Treating `int[] b = a;` as a copy. Every change through `b` changes `a`. For an independent copy, write `int[] b = a.clone();` or `Arrays.copyOf(a, a.length)`.

3.2.7 Ready-made operations: java.util.Arrays

Java arrays have no sorting, searching, or printing methods of their own. Those come as static methods of the `Arrays` class in `java.util`, called through the class name — `Arrays.sort(a)`, not `a.sort()` — after `import java.util.Arrays;`. Table 3.5 lists the common ones.

Method	What it does	Cost
`Arrays.toString(a)`	Produces a string such as `"[1, 2, 3]"`	*O(n)*

Method	What it does	Cost
`Arrays.equals(a, b)`	Are the lengths and the elements, pairwise, equal?	*O(n)*
`Arrays.fill(a, v)`	Writes `v` into every cell	*O(n)*
`Arrays.copyOf(a, n)`	New array; if `n > a.length`, the extra cells get the default value	*O(n)*
`Arrays.sort(a)`	Sorts into ascending order (in place)	*O(n log n)*
`Arrays.binarySearch(a, x)`	Binary search in a sorted array; a negative result if absent	*O(log n)*

Table 3.5: Commonly used methods of `java.util.Arrays`.

Most are *O(n)*, touching each element once. `sort` costs more, *O(n log n)* for a good algorithm (3.5.7). `binarySearch` costs less but needs a sorted array. **Binary search** compares the target with the middle element and continues in the left half if the target is smaller, the right half if larger. The range halves each step, so about $log_2 n$ steps suffice: 20 for a million elements.

The two most common array mistakes show why the class is needed:

```
int[] a = {1, 2, 3};
int[] b = {1, 2, 3};
a == b              // false: different objects
Arrays.equals(a, b) // true: same contents
println(a)          // [I@1b6d3586
```

`a` and `b` are separate arrays with equal contents. `==` compares references — do both variables point to the same object? — so it gives `false`. `Arrays.equals(a, b)` compares the lengths, then the elements one by one, and gives `true`.

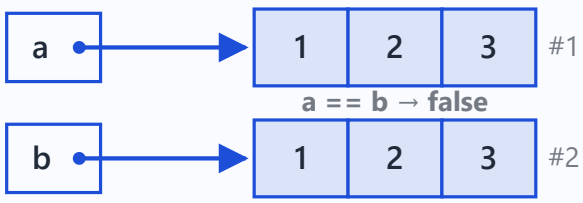


Figure 3.7: `a` and `b` point to two separate array objects (#1 and #2) with the same contents (1, 2, 3). So `a == b` is false and `Arrays.equals(a, b)` is true.

The second mistake is printing an array directly. `System.out.println(a)` prints something like `[I@1b6d3586`: `[I` means "array of `int`," and the part after `@` is an object-specific number that may change between runs. `System.out.println(Arrays.toString(a))` prints `[1, 2, 3]`.

Common mistake

Calling `Arrays.binarySearch` on an unsorted array. There is no error, but the answer may be wrong: searching `{8, 3, 5, 1}` for 8, the method looks at the middle, heads right, and returns −5, "not found," though 8 is the first element. Sort with `Arrays.sort` first.

3.2.8 Variable-length argument lists

A method that averages numbers might be called with two, three, or ten of them. For such cases Java offers **varargs**: three dots after the parameter type.

```
static double average(double... arr) {
    if (arr.length == 0)
        return Double.NaN;  // undefined
    var sum = 0.0;
    for (var value : arr)
        sum += value;
    return sum / arr.length;
}
```

`double... arr` accepts any number of `double` arguments; the compiler packs them into a `double[]` and passes that. Inside, `arr` is an ordinary array with a `length`, traversable by index or for-each. The rules:

- The varargs parameter must come last.
- A method can have only one.
- An existing array can be passed directly: `average(new double[] {1, 2})`.

An empty call, `average()`, is checked separately and returns `Double.NaN` ("Not a Number"), a special `double` marking an undefined result — the average of no numbers is undefined. And `var` declares a local whose type comes from the value on the right: `0.0` is a `double`, so `sum` is too. Table 3.6 shows several calls.

Call	`arr.length`	Result
`average(4, 8, 15)`	3	`9.0`
`average(7.5)`	1	`7.5`
`average()`	0	`NaN`
`average(new double[] {1, 2})`	2	`1.5`

Table 3.6: Several calls of the `average` method.

The idea is familiar: every program's `main(String[] args)` receives its command-line arguments as an array, and `String.format` and `System.out.printf` take variable-length argument lists.

3.2.9 Code trace: average(4, 8, 15)

In `average(4, 8, 15)` the integer arguments are widened to `double`, giving the array [4.0, 8.0, 15.0].

Step	Action	`value`	`sum`
call	the compiler creates the array `{4.0, 8.0, 15.0}`	—	—
`if`	`arr.length` = 3 ≠ 0 → continue	—	—
`var sum`	type inferred as `double`	—	`0.0`
iteration 1	`sum += value`	`4.0`	`4.0`
iteration 2	`sum += value`	`8.0`	`12.0`
iteration 3	`sum += value`	`15.0`	`27.0`
`return`	`27.0 / 3`	—	result `9.0`

Table 3.7: Trace of the call `average(4, 8, 15)`.

Each element is added once: *n* additions, *O(n)* time, and no memory beyond `sum`, `value`, and the argument array. For `average()`, `length` is 0 and line 3 returns `Double.NaN`. Without the check the result would still be `NaN`, since the floating-point division `0.0 / 0` yields it; the explicit check shows the case is handled on purpose.



Figure 3.8: The array the compiler builds for `average(4, 8, 15)`: [4.0, 8.0, 15.0]. Inside the method, `arr` points to it.

3.2.10 Arrays in memory: the stack and the heap

We keep saying “the variable on the stack” and “the array on the heap.” A running Java program divides memory into these two regions, which do different jobs and are managed differently.

- **The stack** holds method calls and local variables. Each call gets a **stack frame** with its parameters and locals, removed automatically when the method returns.
- **The heap** holds every object created with `new`. Every Java array lives there; only its reference is on the stack.

The garbage collector of Section 3.2.3 manages the heap, periodically freeing objects no reference points to. Nobody deletes objects by hand; Java has no command for it.

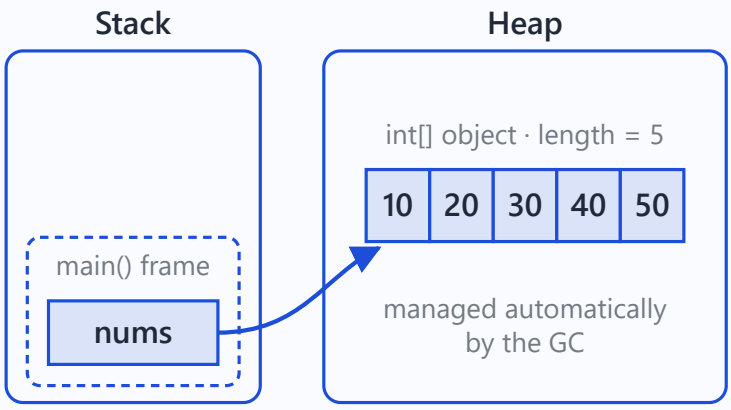


Figure 3.9: The reference `nums` sits in the `main()` frame on the stack and points to a five-element `int[]` on the heap (10, 20, 30, 40, 50; `length` = 5). The garbage collector manages the heap.

	Stack	Heap
What it holds	local variables, references, call frames	objects, arrays
Lifetime	until the method returns	as long as it is reachable
Management	automatic (call / return)	automatic (garbage collector)

Table 3.8: The stack and the heap compared.

The different lifetimes matter: an array created in a method outlives the method. The frame and the local holding the reference disappear, but the array stays on the heap, and if the method returned the reference, the caller keeps using it. `readArray` in Section 3.3.1 works exactly this way.

An array lives while at least one reference to it exists. When the last one goes — the variable gets another array, is set to `null`, or its method returns — the array is unreachable and may be collected.

3.2.11 "Static" and "dynamic" arrays: C and Java

Books often contrast **static arrays** with **dynamic arrays**. The terms fit C and C++ best, where an array can live in two places. Table 3.9 compares the languages.

	C / C++	Java
Array on the stack	`int a[10];` — size fixed at compile time, gone when the function returns	None: every array is on the heap
Array on the heap	`malloc` / `new int[n]`	`new int[n]` (initializer lists also go on the heap)
Size	set at run time on the heap	set at run time, then fixed
Freeing	the programmer: `free` / `delete[]`	the garbage collector, automatically; `arr = null` only drops a reference

Table 3.9: Where arrays live and how long, in C/C++ and Java.

In C, `int a[10];` puts the array in the function's stack frame; its size is known at compile time and it vanishes with the frame. Such arrays are "static." An array whose size is known only at run time is allocated on the heap with `malloc` (C) or `new` (C++) and called "dynamic." In C the programmer must `free` it when done: forgetting leaks memory, and freeing twice is undefined behavior, usually a crash.

Java has no such split. Every array is on the heap, its size computed at run time if need be but fixed afterward:

```
int[] dynArr = new int[n];   // room for n elements on the heap
dynArr = null;               // GC may collect it if no other reference exists
```

Line 1 creates an array of as many elements as `n` currently holds. Line 2 does not delete it; it drops the reference in `dynArr`, and if nothing else points to the array, the GC collects it when convenient. Even `System.gc()` only asks the JVM; it guarantees nothing.

"Dynamic array" also means an array that grows by itself when full, like Java's `ArrayList`: it keeps an ordinary array inside and, when it fills, copies the elements into a larger one. Such a structure is built on a fixed-size array whose own size still never changes.

> **Common mistake**
>
> Believing `arr = null;` deletes the array at once. It drops one reference. If another variable points to the array, it lives on; if none does, the garbage collector still decides when to collect it.

3.2.12 Index out of bounds

Valid indices run from 0 to $n - 1$. An index outside that range — negative, or at least `length` — makes Java throw an `ArrayIndexOutOfBoundsException`, a subclass of `IndexOutOfBoundsException`.

```
int[] arr = new int[5];
arr[5] = 1;  // Index 5 out of
             // bounds for length 5
arr[-1] = 1; // same error
```

There is no cell 5. `arr[5] = 1` compiles, since the compiler usually cannot know an index's value, but the JVM checks every index at run time and throws, with a plain message: `Index 5 out of bounds for length 5`. `arr[-1]` fails the same way. Uncaught, the exception ends the program there.



Figure 3.10: In a five-element array the valid indices are 0 … `length` − 1, that is, 0–4. The dashed cell 5 lies outside the array; accessing `arr[5]` is an error.

This check is a major safety feature that sets Java apart from C. C computes the address of `a[5]` anyway and reads or overwrites whatever follows the array, silently corrupting any variable stored there; the bug may surface only when the program misbehaves somewhere else. Such overflows have caused many security holes. Java reports the error on the line that made it, at the price of one small comparison per access.

An index that comes from the user or a calculation can be checked first:

```
if (index >= 0
        && index < arr.length) {
    int value = arr[index];
} else {
    System.out.println("Invalid");
}
```

The access is safe when both conditions hold: the index is at least 0 and less than `length`.

Common mistake

Writing `i <= a.length` in a loop condition. This **off-by-one error** makes the last iteration access `a[a.length]` and throw. The correct loop is `for (int i = 0; i < a.length; i++)`.

3.3 Array algorithms

These algorithms are small but instructive, and each uses the array differently: one pass, sort first, or work inward from both ends. The surest way to understand one is to trace every step on paper with a small input, as the trace tables below do.

3.3.1 Reading an array from the user

The mean, median, and mode programs all begin by asking how many numbers will be entered and then reading them. That shared work belongs in a helper method:

```
import java.util.Scanner;

static double[] readArray(Scanner scanner) {
    System.out.print("Enter the array size: ");
    int size = scanner.nextInt();
    double[] arr = new double[size];  // size known only at run time
    for (int i = 0; i < size; i++) {
        System.out.print("Enter element " + (i + 1) + ": "); // count from 1
        arr[i] = scanner.nextDouble();
    }
    return arr;
}
```

The `Scanner` is created outside and passed in so the whole program uses one; several scanners on the same input can mix up what they read.

1. The prompt asks for the size; `print` adds no line break, so the cursor waits beside it.
2. `nextInt()` reads the integer into `size`.
3. `new double[size]` creates the array, its size known only now, at run time — no problem in Java. Every cell starts at `0.0`.
4. The loop reads the elements. The user sees `i + 1`: the array counts from 0, people from 1, and "Enter element 0" would confuse them.
5. The array's reference is returned.

The last step is the memory model of Section 3.2.10 at work: the local `arr` vanishes when the method returns, but the array survives on the heap and is used through the reference the caller stores.

Table 3.10 summarizes the three statistics of this section: same array, different costs.

Measure	Definition	Cost
Mean	Sum / number of elements	*O(n)*
Median	Middle of the sorted array	*O(n log n)*
Mode	Most frequent value	*O(n log n)*

Table 3.10: Three statistics computed on an array.

3.3.2 Code trace: readArray(sc)

Suppose the user enters 4 as the size, then 4, 8, 15, and 16 (Table 3.11).

Step	On screen	Read	`arr`
size prompt	Enter the array size:	`size` = 4	not yet created
`new double[4]`	—	—	[0.0, 0.0, 0.0, 0.0]

Step	On screen	Read	`arr`
`i = 0`	Enter element 1:	4	[4.0, 0.0, 0.0, 0.0]
`i = 1`	Enter element 2:	8	[4.0, 8.0, 0.0, 0.0]
`i = 2`	Enter element 3:	15	[4.0, 8.0, 15.0, 0.0]
`i = 3`	Enter element 4:	16	[4.0, 8.0, 15.0, 16.0]
`i = 4`	4 < 4 false → loop ends	—	[4.0, 8.0, 15.0, 16.0]
`return`	—	—	reference returned

Table 3.11: Trace of `readArray(sc)`. The user enters 4, 4, 8, 15, 16.

The array does not exist until the user types the size (Figure 3.11), and `nextDouble()` reads the user's 4 as `4.0`, since the array holds `double`s.

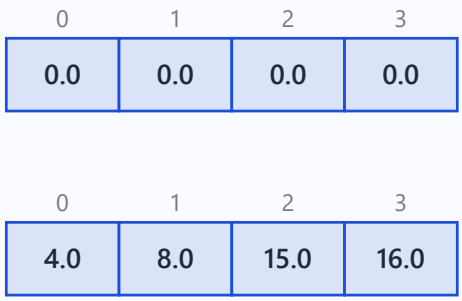


Figure 3.11: The array during `readArray`. Left, right after `new double[4]`: four cells of `0.0`. Right, when the loop ends: 4.0, 8.0, 15.0, 16.0.

Reading *n* elements takes *n* reads, *O(n)*. In practice most of the time goes to waiting for the user, but the algorithm's work is proportional to the element count.

Common mistake

Making the index match the number the user sees. A loop from `i = 1` to `i <= size` shows the right numbers but leaves `arr[0]` unused and finally accesses `arr[size]`, out of bounds. Count the index from 0 and display `i + 1`.

3.3.3 Computing the mean

The mean is the sum of the values divided by their number. This `main` reads the array with `readArray`, computes the average, and prints it:

```
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    double[] arr = readArray(sc);
    double sum = 0;
    for (int i = 0; i < arr.length; i++)
        sum += arr[i];
    double average = sum / arr.length;
    System.out.println("Array: "
        + Arrays.toString(arr));
    System.out.println("Average: "
        + average);
    sc.close();
}
```

At its heart is an **accumulator**: `sum` starts at 0 and each iteration adds an element. The same pattern underlies finding a maximum, counting elements that meet a condition, or computing a product.

`sum` is a `double` initialized with the integer `0`, which Java widens to `0.0`; the division is done in `double`, so the result keeps its fraction. `Arrays.toString(arr)` formats the array — `println(arr)` would print something like `[D@…` (Section 3.2.7) — and `sc.close()` closes the `Scanner`.

Example 3.3

This time the user enters 5 values, and `readArray` returns {4, 8, 15, 16, 23}.

1. `sum` = 0.0.
2. `i = 0` : 0.0 + 4.0 = 4.0.
3. `i = 1` : 4.0 + 8.0 = 12.0.
4. `i = 2` : 12.0 + 15.0 = 27.0.
5. `i = 3` : 27.0 + 16.0 = 43.0.
6. `i = 4` : 43.0 + 23.0 = 66.0.
7. `i = 5` : 5 < 5 is false; the loop ends.
8. `average` = 66.0 / 5 = 13.2.

The program prints `Array: [4.0, 8.0, 15.0, 16.0, 23.0]` and `Average: 13.2`.



Figure 3.12: The array of Example 3.3: 4.0, 8.0, 15.0, 16.0, 23.0. The sum is 66.0 and the mean 66.0 / 5 = 13.2.

n elements take *n* additions, *O(n)* time. Beyond the array the algorithm uses only `sum`, `i`, and `average`, however large *n* grows, so the extra memory is *O(1)*.

Consider the edge case. A size of 0 creates a valid, empty array; the loop never runs and `0.0 / 0` is computed, a floating-point division that raises no error and yields `NaN`. "Average: NaN" is technically right but unhelpful; a warning such as "no numbers entered" would be better.

Common mistake

Keeping the sum and count as integers. `int` division truncates: `66 / 5` gives 13, not 13.2, and for an empty array the integer division `0 / 0` throws an `ArithmeticException`. Make at least one operand a `double`.

3.3.4 The median

The mean is very sensitive to extreme values. If five people earn 3, 5, 7, 9, and 1000 units a month, the mean is (3 + 5 + 7 + 9 + 1000) / 5 = 204.8, a figure that describes none of them. The income of the person in the middle says more.

Definition 3.3

The **median** of an array of numbers is the value that falls in the middle when the array is sorted. If the number of elements *n* is odd, the median is the element at index *n / 2* of the sorted array (with integer division). If *n* is even, two elements share the middle, at indices *n/2 − 1* and *n/2*, and the median is their mean.

Sorted, the incomes are 3, 5, 7, 9, 1000, and the median is 7 — still 7 if 1000 became 10 or a million. Extreme values do not move the median.

```java
static double median(double[] arr) {
    double[] s = arr.clone(); // copy
    Arrays.sort(s);             // O(n log n)
    int size = s.length;
    double median;
    if (size % 2 == 0) {       // even
        int m1 = size/2 - 1, m2 = size/2;
        median = (s[m1] + s[m2]) / 2;
    } else {                    // odd
        median = s[size / 2];
    }
    return median;
}
```

1. **Copy.** `arr.clone()` copies the array, because the next step sorts and sorting `arr` itself would scramble the caller's array (3.2.6). Asking for a median should not rearrange your data.
2. **Sort.** `Arrays.sort(s)` sorts the copy ascending, in *O(n log n)*.
3. **Odd or even?** `size % 2` is the remainder on division by 2; 0 means even.
4. **The middle value.** For an even size, average the values at `m1 = size/2 - 1` and `m2 = size/2`; for an odd size, return `s[size / 2]`, where integer division gives exactly the middle index.

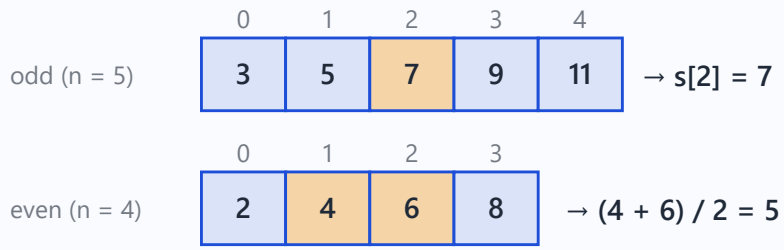


Figure 3.13: The two cases of the median. Top, odd size ($n = 5$): in the sorted array 3, 5, 7, 9, 11 the middle element is `s[2]` = 7. Bottom, even size ($n = 4$): in 2, 4, 6, 8 the median is the mean of the two middle values, (4 + 6) / 2 = 5.

3.3.5 Code trace: median()

We call the method on an even-sized and an odd-sized array: {6, 2, 8, 4}, then {9, 3, 11, 7, 5}.

Step	Call 1: {6, 2, 8, 4}	Call 2: {9, 3, 11, 7, 5}
`s = arr.clone()`	[6.0, 2.0, 8.0, 4.0]	[9.0, 3.0, 11.0, 7.0, 5.0]
`Arrays.sort(s)`	[2.0, 4.0, 6.0, 8.0]	[3.0, 5.0, 7.0, 9.0, 11.0]
`size`	4	5
`size % 2`	0 → even	1 → odd
middle index(es)	`m1` = 1, `m2` = 2	5 / 2 = 2

Step	Call 1: {6, 2, 8, 4}	Call 2: {9, 3, 11, 7, 5}
computation	(4.0 + 6.0) / 2	`s[2]`
`return`	`5.0`	`7.0`
caller's array	unchanged: [6.0, 2.0, 8.0, 4.0]	unchanged: [9.0, 3.0, 11.0, 7.0, 5.0]

Table 3.12: Trace of two calls of `median`.

The last row shows the copy paying off: only `s` was sorted, and the callers' arrays keep their order.

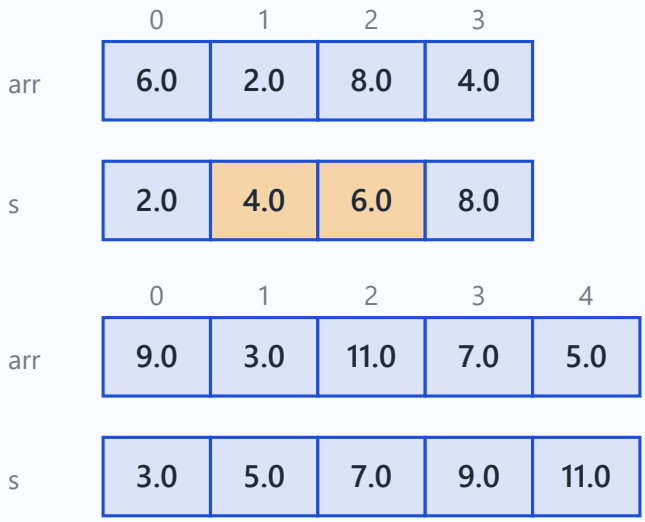


Figure 3.14: Final state of the two calls. Left, call 1: the caller's array (6, 2, 8, 4) above the sorted copy `s`, with the middle values 4.0 and 6.0 highlighted. Right, call 2: 9, 3, 11, 7, 5 and its sorted copy 3, 5, 7, 9, 11.

Sorting decides the cost: copying is *O(n)*, sorting *O(n log n)*, reading the middle *O(1)*, so the total is the largest term, *O(n log n)*. The copy needs *O(n)* extra memory. Advanced algorithms find the median without sorting in *O(n)* average time, but sorting is the easiest route to understand.

Common mistake

Taking the middle of the unsorted array. For {6, 2, 8, 4} the middle elements 2 and 8 average to 5, right by coincidence; for {9, 3, 11, 7, 5} the middle element is 11 but the median is 7. The median is the middle of the *sorted* array.

3.3.6 The mode

Definition 3.4

The **mode** of an array is its most frequent value. If several values tie for the highest count, the array has several modes, and an algorithm must pick one of them.

Counting each value's occurrences by rescanning the whole array costs $n \times n$ comparisons. Sorting first is better: equal values end up side by side, so one left-to-right pass counts every run.

```
static int mode(int[] arr) { // not empty
    int[] s = arr.clone();
    Arrays.sort(s);  // equal values side by side
    int best = s[0], bestCount = 1;
    int count = 1;
    for (int i = 1; i < s.length; i++) {
        if (s[i] == s[i - 1]) count++;
        else count = 1;   // new value
        if (count > bestCount) {
            bestCount = count;
            best = s[i];
        }
    }
    return best;
}
```

- `s` : a sorted copy, made for the same reason as in `median` .
- `count` : how many times in a row the current value has appeared so far.
- `best` , `bestCount` : the record — value and length of the longest run so far.

Initially `s[0]` has been seen once and holds the record. From index 1, each element is compared with its left neighbor: equal means the run continues and `count` grows; different means a new value, and `count` resets to 1. After each comparison, a count that beats the record replaces it.

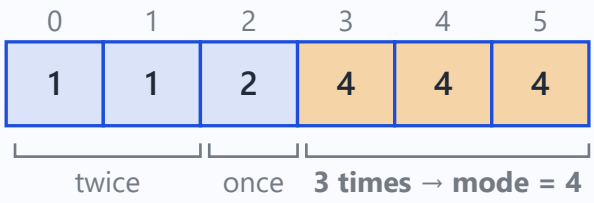


Figure 3.15: The sorted array 1, 1, 2, 4, 4, 4. Equal values sit together: 1 occurs twice, 2 once, 4 three times. The mode is 4.

Using `>` is deliberate. A value that repeats as often as the record only *ties* it and cannot *beat* it, so the record stays with the value seen first — in an ascending array, the smallest of the tied values. With `>=`, ties would go to the largest.

The comment "not empty" is a precondition: an empty array has no `s[0]`, and the method would throw an `ArrayIndexOutOfBoundsException`. The caller must not pass one.

3.3.7 Code trace: mode()

In `mode({4, 1, 4, 2, 1, 4})` the equal values are scattered; the sorted copy is `s` = {1, 1, 2, 4, 4, 4}. Initially `best` = 1, `bestCount` = 1, `count` = 1.

`i`	`s[i]` vs. `s[i-1]`	`count`	Record beaten?	`best` / `bestCount`
1	1 == 1 → same	2	2 > 1 yes	1 / 2
2	2 ≠ 1 → new value	1	1 > 2 no	1 / 2
3	4 ≠ 2 → new value	1	1 > 2 no	1 / 2
4	4 == 4 → same	2	2 > 2 no	1 / 2
5	4 == 4 → same	3	3 > 2 yes	4 / 3

Table 3.13: Trace of `mode({4, 1, 4, 2, 1, 4})`; `s` = {1, 1, 2, 4, 4, 4}.

The method returns 4, which occurs three times. Row `i = 4` shows the tie rule: 4 has appeared twice, matching the record, so `2 > 2` fails and 1 keeps it. Had the array ended there, the method would return 1, seen first; here 4 appears once more and takes the record at `i = 5`.

Sorting costs *O(n log n)* and the scan *O(n)*, so the total is *O(n log n)*, with *O(n)* extra memory for the copy.

> **Common mistake**
>
> Deleting the sort as unnecessary. The algorithm compares only *neighbors*, and in the unsorted {4, 1, 4, 2, 1, 4} no neighbors are equal: `count` stays 1 and the method returns the first element, 4 — right here by coincidence, wrong whenever the first element is not the mode. Sorting brings equal values together, as the algorithm assumes.

3.3.8 Reversing an array in place

Reversing an array moves the first element to the end, the last to the front, the second to the second-to-last place, and so on: {3, 8, 1, 9, 4, 6} becomes {6, 4, 9, 1, 8, 3}. Copying back to front into a new array, `result[i] = arr[n - 1 - i]`, works but needs a second array of *n* elements. The job can be done within the array itself.

An algorithm that works on the array itself, without an extra array, is **in-place**. Here: exchange the first and last elements, then the second and second-to-last, and so on to the middle. Exchanging two elements is a **swap**.

```
static void reverse(int[] arr) {
    int start = 0;
    int end = arr.length - 1;
    while (start < end) {
        // swap the two outermost elements
        int temp = arr[start];
        arr[start] = arr[end];
        arr[end] = temp;
        start++;   // move inward
        end--;
    }
}
```

The indices `start` and `end` begin at the two ends; since they "point at" positions, they are also called **pointers**. Each iteration swaps the elements they point to and moves both one step inward. When they meet, `start < end` fails and the loop ends.

A swap needs three assignments and a temporary. The first saves the left value in `temp`; the second copies the right value over it, so the old left value now exists only in `temp`; the third writes it to the right. Without `temp`, `arr[start] = arr[end]; arr[end] = arr[start];` would leave both cells equal and lose the left value.

Iteration	`start`, `end`	Condition	`temp`	`arr` after swap
1	0, 5	0 < 5	3	[6, 8, 1, 9, 4, 3]
2	1, 4	1 < 4	8	[6, 4, 1, 9, 8, 3]
3	2, 3	2 < 3	1	[6, 4, 9, 1, 8, 3]
—	3, 2	3 < 2 false → ends	—	[6, 4, 9, 1, 8, 3]

Table 3.14: Trace of `reverse({3, 8, 1, 9, 4, 6})`.



Figure 3.16: Three moments of the reversal. (a) Start: `start` at index 0; the array is 3, 8, 1, 9, 4, 6. (b) After the first swap and a step inward: 6, 8, 1, 9, 4, 3, with `start` = 1, `end` = 4 and 3 last saved in `temp`. (c) End: the pointers have crossed (`end` = 2, `start` = 3); the array is 6, 4, 9, 1, 8, 3.

Six elements took three swaps. With an odd count the middle stays put: in {1, 2, 3, 4, 5}, 1 swaps with 5, 2 with 4, and 3 is untouched, giving {5, 4, 3, 2, 1}.

Theorem 3.1 (Cost of reversal)

The `reverse` method reverses an array of n elements with $\lfloor n/2 \rfloor$ swaps, in $O(n)$ time and $O(1)$ extra memory.

Proof. Each swap puts one left and one right element in place and brings the pointers 2 closer. Initially `end - start = n - 1`, and the loop runs while this is positive. For even n it reaches −1 after $n/2$ iterations; for odd n it reaches 0 after $(n - 1)/2$, leaving the middle element in place. Either way there are $\lfloor n/2 \rfloor$ iterations, each doing constant work (three assignments, an increment, a decrement, a comparison), so the time is $O(n)$. Only `start`, `end`, and `temp` are used, so the extra memory is $O(1)$. □

Copying into a new array also takes $O(n)$ time; the difference is memory, $O(n)$ extra versus $O(1)$. With tight memory or a huge array, reverse in place; if the original is still needed elsewhere, copying has the advantage of leaving it intact.

Common mistake

Looping all the way, swapping `arr[i]` and `arr[n-1-i]` for every `i` up to `n - 1`. The first half reverses the array and the second half swaps the same pairs back. Stop in the middle, after *n/2* iterations.

3.3.9 Application: the Caesar cipher

The **Caesar cipher**, named after Julius Caesar, replaces every letter with the one a fixed *k* places further along the alphabet, wrapping around at the end. With *k* = 3, A → D, B → E, C → F, and X → A, Y → B, Z → C. With only 25 keys, all easy to try, it offers no security, but it is a fine exercise in character arrays and the remainder operator.

```
static String encrypt(String message, int k) {
    char[] chars = message.toCharArray();
    for (int i = 0; i < chars.length; i++) {
        int pos = chars[i] - 'A';        // 0..25 (A–Z only)
        chars[i] = (char) ('A' + (pos + k) % 26);
    }
    return new String(chars);
}
```

- `toCharArray()`. Java `String`s are immutable, so the letters are copied into a `char[]`, whose elements can be changed in place.
- `chars[i] - 'A'`. A `char` is an integer code, and the capital letters have consecutive codes, so subtracting `'A'` gives a letter's alphabet position: 0 for `'A'`, 4 for `'E'`, 25 for `'Z'`.
- `(pos + k) % 26`. The position advances by *k*; past 25, the remainder wraps it around: *(25 + 3) % 26 = 2*.
- `(char) ('A' + …)`. Adding the position to `'A'` and casting gives the letter back.
- `new String(chars)`. The modified array becomes a string again.

Tracing `encrypt("ZEKA", 3)`:

`i`	`chars[i]`	`pos`	`(pos + 3) % 26`	**New letter**	`chars`
0	Z	25	28 % 26 = 2	C	[C, E, K, A]
1	E	4	7	H	[C, H, K, A]
2	K	10	13	N	[C, H, N, A]
3	A	0	3	D	[C, H, N, D]

Table 3.15: Trace of `encrypt("ZEKA", 3)`.

The method returns `"CHND"`. The first row shows the wraparound: Z is at 25, three steps on is 28, and 28 % 26 = 2 is C.

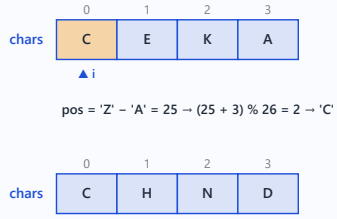


Figure 3.17: Two moments of the Caesar cipher. Left, `i = 0`: `pos` = 'Z' − 'A' = 25, (25 + 3) % 26 = 2 → 'C', so the first cell is now C. Right, when the loop ends, `chars` is C, H, N, D.

Decryption needs no new method: 3 steps forward plus 23 more make 26, a full trip around the alphabet, so calling the same method with *k = 26 − 3 = 23* decrypts: `encrypt("CHND", 23)` gives `"ZEKA"`.

Each letter is processed once, so *n* letters take *O(n)*; `toCharArray` and `new String` add one *O(n)* copy each, which does not change the total.

> **Common mistake**
>
> Assuming the method handles any text. It assumes capital letters A–Z only; for spaces, digits, or lowercase letters, `chars[i] - 'A'` falls outside 0–25 and the output is gibberish. Turkish letters such as Ç, Ğ, İ, Ö, Ş, Ü also lie outside A–Z. A real application should leave such characters unchanged or handle them with a separate alphabet array.

3.4 Multidimensional arrays

Some data naturally forms a table: a weekly timetable, a chessboard, the pixels of an image, a table of distances between neighborhoods. It can be squeezed into a one-dimensional array, but then every access must turn a row and a column into one index. A two-dimensional array leaves that to the language.

3.4.1 Two-dimensional arrays

A **two-dimensional array** arranges elements in rows and columns, like a table; it is programming's version of the mathematical **matrix**. An element takes two indices, `matrix[row][col]`, row first, both starting at 0.

```
int[][] matrix = new int[3][3];
matrix[1][2] = 7;   // 2nd row, 3rd column
```

Line 1 creates a 3 × 3 matrix of zeros; line 2 writes 7 into column 2 of row 1 — counting the human way, the third column of the second row.

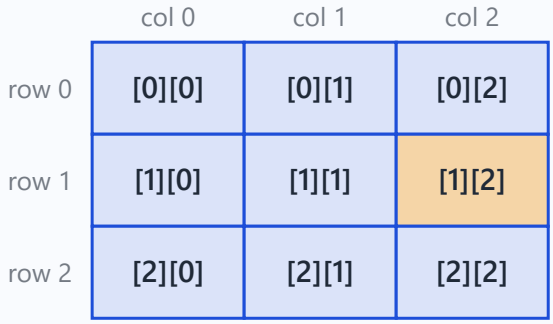


Figure 3.18: A matrix of 3 rows and 3 columns. Each cell is addressed as `[row][col]`; the highlighted cell is `[1][2]`, column 2 (the third column) of row 1 (the second row).

Two-dimensional arrays suit chessboards (8 × 8), images, and spreadsheets. In an image, `image[y][x]` is the color of the pixel in row `y` from the top and column `x` from the left.

> **Common mistake**
>
> Mixing up the index order. `matrix[1][2]` and `matrix[2][1]` are different cells, and the first index always selects the row. Habit from the coordinate plane, *x* (horizontal) before *y* (vertical), leads people to write `image[x][y]` and swap rows with columns.

3.4.2 Matrices in Java: arrays of arrays

Some languages store a two-dimensional array as one block, row after row. Java has no true two-dimensional type: a matrix is an **array of arrays**, and `int[][]` is an array whose elements are `int[]`.

```
int[][] matrix = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
matrix.length      // 3 (number of rows)
matrix[0].length   // 3 (number of columns)
```

This creates four arrays: an **outer array** of three references, and three `int` arrays, one per row. Each outer cell points to a row, and `matrix` points to the outer array.

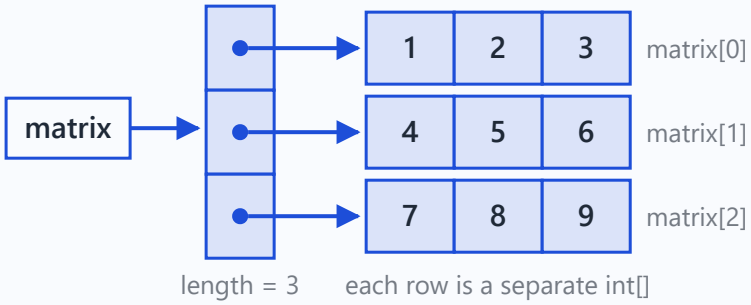


Figure 3.19: `matrix` points to an outer array of 3 references (`length` = 3), each pointing to a separate `int[]` row: `matrix[0]` = 1 2 3, `matrix[1]` = 4 5 6, `matrix[2]` = 7 8 9.

- `matrix.length` is the outer array's length, the **number of rows** (3).
- `matrix[0]` is a row, and `matrix[0].length` its length, the **number of columns** (3).
- `matrix[1][2]` takes two steps: `matrix[1]` reaches row 1, then element 2 of that row is read: 6.
- The initializer states no sizes; the inner lists determine them.

Most important, each row is a separate array, so rows may differ in length. Such an array is a **jagged array**.

3.4.3 A jagged array: Pascal's triangle

Definition 3.5

A **jagged array** is a two- (or higher-) dimensional array whose rows may differ in length. In Java the outer array is created first, with only the number of rows: `new int[n][]` . Each row is then created separately, with its own length: `arr[i] = new int[k];`

Pascal's triangle is a natural example. Each row is one longer than the last, both ends are 1, and each inner number is the sum of the two above it:

> $p[i][j] = p[i-1][j-1] + p[i-1][j]$

The first four rows are 1; 1 1; 1 2 1; 1 3 3 1. A square matrix would give every row the room of the longest and leave nearly half its cells empty; a jagged array gives each row just what it needs. This code builds 4 rows:

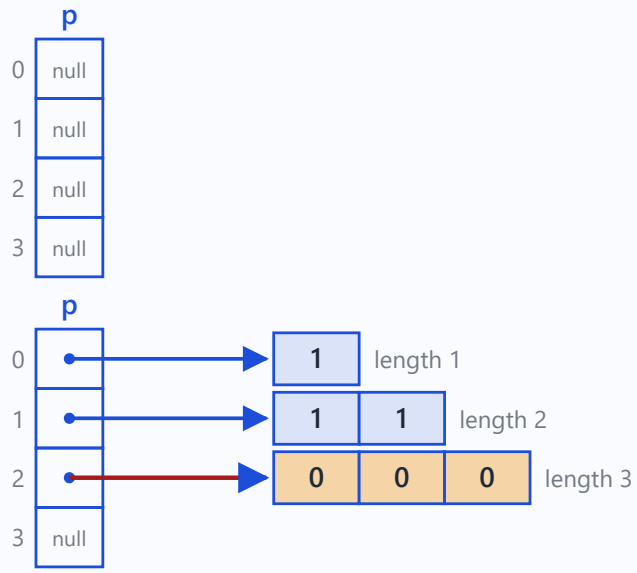
```
int n = 4;
int[][] p = new int[n][];          // no rows yet
for (int i = 0; i < n; i++) {
    p[i] = new int[i + 1];         // length i + 1
    p[i][0] = p[i][i] = 1;
    for (int j = 1; j < i; j++)
        p[i][j] = p[i-1][j-1] + p[i-1][j];
}
```

`new int[n][]` creates only the outer array; the empty second brackets mean no rows yet, so all four cells are `null`. Each iteration creates row `i` with length `i + 1` and sets both ends to 1. `p[i][0] = p[i][i] = 1` is a **chained assignment**, evaluated right to left: `p[i][i] = 1` first, then its value, 1, goes into `p[i][0]`. The inner loop computes the cells between the ends from the row above.

`i`	Row created	Ends	Inner cells (`j` = 1 … `i` − 1)	`p[i]`
—	`new int[4][]`	—	—	all four rows `null`
0	`new int[1]`	`p[0][0]` = 1	none (`j` < 0)	[1]
1	`new int[2]`	`p[1][0]` = `p[1][1]` = 1	none (`j` < 1)	[1, 1]
2	`new int[3]`	`p[2][0]` = `p[2][2]` = 1	`p[2][1]` = 1 + 1 = 2	[1, 2, 1]
3	`new int[4]`	`p[3][0]` = `p[3][3]` = 1	`p[3][1]` = 1 + 2 = 3; `p[3][2]` = 2 + 1 = 3	[1, 3, 3, 1]

Table 3.16: Building Pascal's triangle (`n` = 4).

In row 0, `p[0][0]` is both first and last cell, so it gets 1 twice. Rows 0 and 1 have no inner cells, so the inner loop never runs. Row 3 comes from row 2: `p[3][1] = p[2][0] + p[2][1] = 1 + 2 = 3` and `p[3][2] = p[2][1] + p[2][2] = 2 + 1 = 3`.



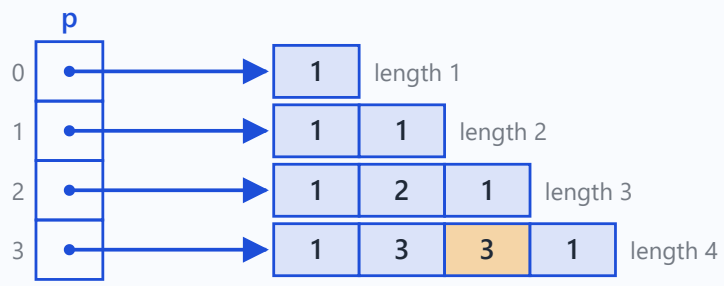


Figure 3.20: Building Pascal's triangle. (a) After `new int[4][]`: all four outer cells are `null`. (b) Right after `p[2] = new int[3]`: two rows filled, the new row still zeros. (c) Final state: four rows of lengths 1, 2, 3, 4, with the last computed cell, `p[3][2]` = 3, highlighted.

The triangle uses 1 + 2 + 3 + 4 = 10 cells, a 4 × 4 matrix 16. For n rows it is $n(n + 1)/2$ against n^2, saving nearly half the memory.

Common mistake

Using rows before creating them. After `int[][] m = new int[3][];`, `m[0][0] = 5` throws a `NullPointerException`, because `m[0]` is still `null`. Give each row an array first: `m[i] = new int[k];`.

3.4.4 Shallow and deep copies of a 2D array

Being an array of arrays, a matrix copies in a surprising way. Section 3.2.6 said `clone()` makes an independent copy, true for an array of nums. But the outer array's elements are references, and `clone()` copies the references, not the rows.

Definition 3.6

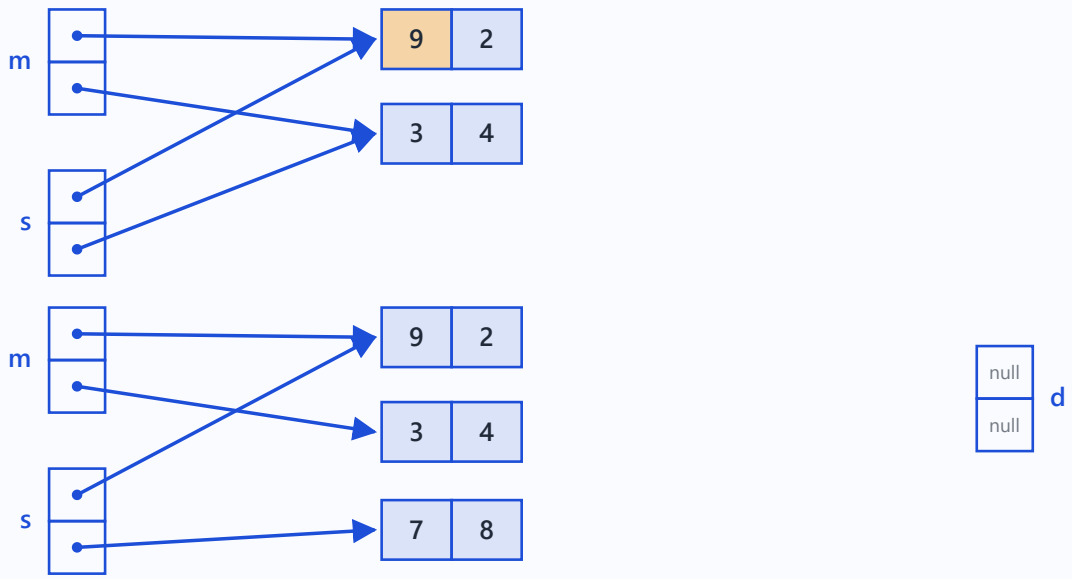
A **shallow copy** of an array of references copies only the array itself; the copy's elements point to the same objects as the original's. A **deep copy** copies the referenced objects too, so the copy and the original share no objects.

The two kinds side by side:

```
int[][] m = { {1, 2}, {3, 4} };
int[][] s = m.clone();          // shallow copy
s[0][0] = 9;                    // m[0][0] is 9 too!
s[1] = new int[] {7, 8};        // only s changes
int[][] d = new int[2][];       // deep copy:
for (int i = 0; i < 2; i++)
    d[i] = m[i].clone();        // rows too
d[0][1] = 5;                    // m unaffected
```

Example 3.4

1. `int[][] m = { {1, 2}, {3, 4} };` The outer array `m` points to the rows {1, 2} and {3, 4}.
2. `int[][] s = m.clone();` Only the outer array is copied: `s` is a new outer array holding the same two references, so `s[0]` and `m[0]` are the same row. A shallow copy.
3. `s[0][0] = 9;` The change lands in the shared row, so `m[0][0]` is 9 too: `m` = {{9, 2}, {3, 4}}.
4. `s[1] = new int[] {7, 8};` This changes a reference in `s`'s own outer array, not a row's contents. `s[1]` now points to a new row; `m[1]` is still {3, 4}. `s` = {{9, 2}, {7, 8}}.
5. `int[][] d = new int[2][];` A deep copy starts with a new outer array of two references and no rows.
6. **Loop:** `d[0] = m[0].clone()` and `d[1] = m[1].clone()` copy each row. With `m[0]` now {9, 2}, `d[0]` = {9, 2} and `d[1]` = {3, 4}.
7. `d[0][1] = 5;` Only `d`'s own row changes: `d` = {{9, 5}, {3, 4}}, while `m[0][1]` is still 2.



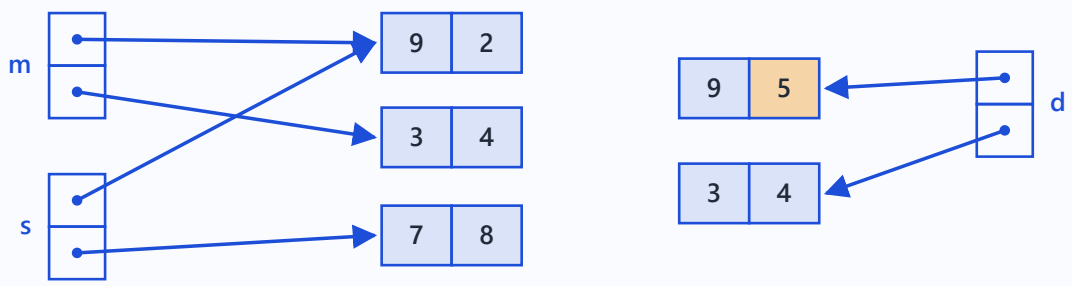


Figure 3.21: Shallow and deep copies. (a) After `s[0][0] = 9`: the outer arrays `m` and `s` point to the same two rows, so a change in a shared row shows through both. (b) After `s[1] = new int[] {7, 8}` and `d = new int[2][]`: `s[1]` points to a separate row; both cells of `d` are `null`. (c) Final state: `d`'s rows (9, 5 and 3, 4) are independent of `m`'s.

Steps 3 and 4 are the key. `s[0][0] = …` changes the *inside* of a shared row, visible through `m`; `s[1] = …` changes a *reference* in `s`'s separate outer array, leaving `m` alone. In a shallow copy, the outer arrays are separate and the rows shared.

A deep copy copies every cell: $O(r \cdot c)$ time and extra memory for r rows and c columns. A shallow copy takes $O(r)$; cheap, but safe only if the rows never change.

> **Common mistake**
>
> Copying a matrix with `m.clone()` or `Arrays.copyOf(m, m.length)` and then changing cells of the copy. Both copies are shallow, so the changes reach the original. Copy each row too: `d[i] = m[i].clone();`

3.4.5 Filling and printing a matrix

The standard way to visit every cell is two nested loops, rows outside, columns inside. This program fills a 3 × 3 matrix with 1 to 9 and prints it as a table:

```
public class MultiArrayExample {
  public static void main(String[] a) {
    int[][] matrix = new int[3][3];
    // initial values: 1…9
    for (int i = 0; i < 3; i++)      // row
      for (int j = 0; j < 3; j++)    // column
        matrix[i][j] = i * 3 + j + 1;

    System.out.println("Matrix contents:");
    for (int i = 0; i < 3; i++) {
      for (int j = 0; j < 3; j++)
        System.out.print(matrix[i][j] + " ");
      System.out.println();   // new line
    }
  }
}
```

Output:

```
Matrix contents:
1 2 3
4 5 6
7 8 9
```

`i * 3 + j + 1` numbers the cells row by row: row `i` is preceded by `i` full rows, `i × 3` cells, then `j` more within the row, plus 1 because numbering starts at 1. Table 3.17 lists the nine assignments.

`i`	`j`	`i * 3 + j + 1`	**Assignment**
0	0	0 + 0 + 1	`matrix[0][0]` = 1
0	1	0 + 1 + 1	`matrix[0][1]` = 2
0	2	0 + 2 + 1	`matrix[0][2]` = 3
1	0	3 + 0 + 1	`matrix[1][0]` = 4
1	1	3 + 1 + 1	`matrix[1][1]` = 5
1	2	3 + 2 + 1	`matrix[1][2]` = 6
2	0	6 + 0 + 1	`matrix[2][0]` = 7
2	1	6 + 1 + 1	`matrix[2][1]` = 8
2	2	6 + 2 + 1	`matrix[2][2]` = 9

Table 3.17: Trace of the fill loop. Each row is one run of the inner loop body.

`j` runs through 0, 1, 2 while `i` holds still; then `i` advances and `j` starts over. Every outer iteration contains all the inner ones, so the body runs 3 × 3 = 9 times — $r \cdot c$ times for an $r \times c$ matrix, a cost of $O(r \cdot c)$.

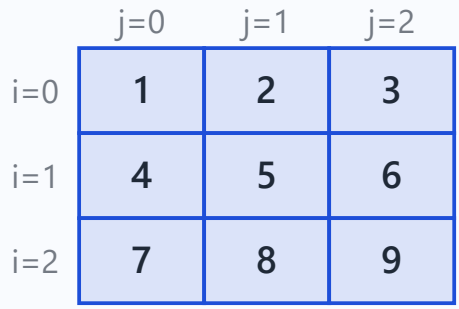


Figure 3.22: The filled matrix. Rows `i` = 0, 1, 2; columns `j` = 0, 1, 2; cells numbered 1 to 9 row by row.

Printing follows the same pattern: the inner loop prints a row on one line with `print`, and the `println()` after it ends the line. Inside the inner loop it would put each number on its own line; without it, all nine would share one.

The bounds here are the constant 3. With `i < matrix.length` and `j < matrix[i].length` the code works for any matrix, even a jagged one, since `matrix[i].length`, unlike `matrix[0].length`, uses each row's own length.

3.4.6 Three-dimensional arrays

Java allows three or more dimensions too. A three-dimensional array is an array of matrices — say, a table changing over time (time × row × column), or the red, green, and blue layers of a color image.

```
int[][][] array3D =
        new int[3][3][3];
for (int x = 0; x < 3; x++)
  for (int y = 0; y < 3; y++)
    for (int z = 0; z < 3; z++)
      array3D[x][y][z] =
          x * 9 + y * 3 + z + 1;

// printing: the same three loops;
// println after the z loop (y),
// println after the y loop (x)
```

Each dimension adds a nested loop; here the innermost assignment runs 27 times. `x * 9 + y * 3 + z + 1` extends the matrix formula by one dimension — 9 cells per layer, 3 per row — so `array3D[1][2][0]` = 1 · 9 + 2 · 3 + 0 + 1 = 16.

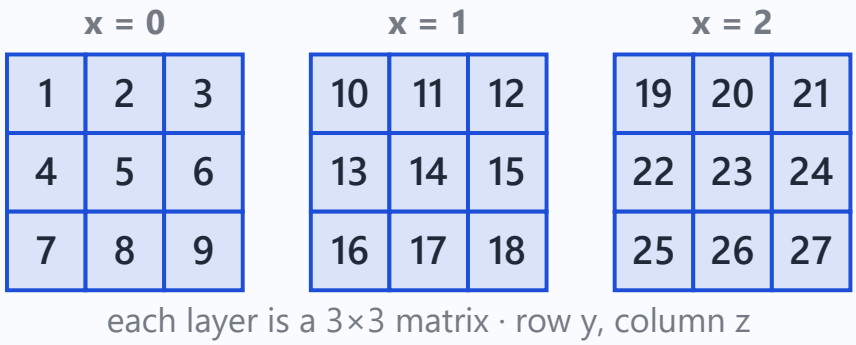


Figure 3.23: A 3 × 3 × 3 array as three layers: `x` = 0 holds 1–9, `x` = 1 holds 10–18, `x` = 2 holds 19–27. Each layer is a 3 × 3 matrix, its row chosen by `y` and column by `z`.

Printing uses the same three loops: when the `z` loop ends a row is complete and `println` starts a new line; when the `y` loop ends a layer is complete and one more blank line follows.

Element counts grow fast with dimensions: a three-dimensional `int` array of 100 per dimension holds a million elements, about 4 MB; four dimensions make it 400 MB. Work out the memory before creating one.

Java allows non-ASCII letters in identifiers, such as the Turkish `üçBoyutluDizi` (“three-dimensional array”), but files may be opened with other encodings and some tools mangle such characters, so ASCII names like `array3D` are safer.

3.4.7 Transposing a matrix

Definition 3.7

The **transpose** of an $m \times n$ matrix M is the $n \times m$ matrix T whose rows are the columns of M: $T[j][i] = M[i][j]$ for all i, j.

Rows become columns and columns rows: a 2 × 3 matrix becomes 3 × 2, and row 0 becomes column 0.

```
static int[][] transpose(int[][] m) {
    int rows = m.length;
    int cols = m[0].length;
    int[][] t = new int[cols][rows];
    for (int i = 0; i < rows; i++)
        for (int j = 0; j < cols; j++)
            t[j][i] = m[i][j];
    return t;
}
static void printMatrix(int[][] m) {
    for (int[] row : m) {
        for (int x : row)
            System.out.print(x + " ");
        System.out.println();
    }
}
```

`transpose` takes the row count from `m.length` and the column count from the first row, `m[0].length`, creates `new int[cols][rows]` with the dimensions swapped, and visits each cell of `m` once, writing it to its mirror position: `t[j][i] = m[i][j]`.

`printMatrix` uses two for-each loops, `row` over the rows and `x` over each row's elements; without indices, it handles jagged arrays too.

Both methods are `static` because `main`, which calls them, is static, and a static method can call its class's other static methods directly, with no object. Without `static`, the call would not compile.

Let's trace `transpose(m)` for `m` = {{1, 2, 3}, {4, 5, 6}}: `rows` = 2, `cols` = 3, and `t` starts as a 3 × 2 matrix of zeros.

`i`	`j`	Assignment	`t` (row by row)
—	—	`new int[3][2]`	[0, 0] [0, 0] [0, 0]
0	0	`t[0][0]` = `m[0][0]` = 1	[1, 0] [0, 0] [0, 0]
0	1	`t[1][0]` = `m[0][1]` = 2	[1, 0] [2, 0] [0, 0]
0	2	`t[2][0]` = `m[0][2]` = 3	[1, 0] [2, 0] [3, 0]
1	0	`t[0][1]` = `m[1][0]` = 4	[1, 4] [2, 0] [3, 0]
1	1	`t[1][1]` = `m[1][1]` = 5	[1, 4] [2, 5] [3, 0]
1	2	`t[2][1]` = `m[1][2]` = 6	[1, 4] [2, 5] [3, 6]

Table 3.18: Trace of `transpose({{1, 2, 3}, {4, 5, 6}})`.

After three assignments, column 0 of `t` is 1, 2, 3, row 0 of `m`; the last three write row 1 into column 1. `printMatrix(t)` prints "1 4," "2 5," and "3 6."

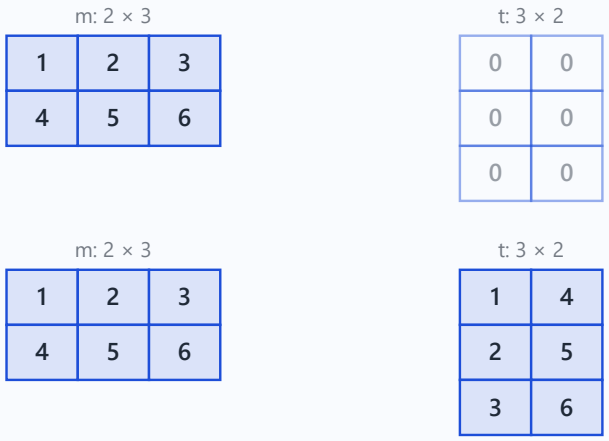


Figure 3.24: Start and end of the transpose trace. Left, `m` (2 × 3) and the new zero-filled `t` (3 × 2). Right, `t` after all assignments: rows 1 4, 2 5, 3 6.

Each element is copied once: $m \cdot n$ assignments, $O(m \cdot n)$ time, and, for the new matrix, $O(m \cdot n)$ extra memory. A square matrix can be transposed in place by swapping each cell above the diagonal with its mirror below; a non-square one cannot, since its shape changes.

Common mistake

Creating the transpose as `new int[rows][cols]`, the original's shape. A square matrix hides the bug; for a 2 × 3 matrix, assigning `t[2][0]` throws an `ArrayIndexOutOfBoundsException`. Another bug is visiting every cell in an in-place transpose: each pair is swapped twice and the matrix returns to its original state. Visit only one side of the diagonal.

3.4.8 Application: Tic Tac Toe

Board games are a classic use of two-dimensional arrays, and Tic Tac Toe is the simplest:

- The board is 3 × 3, stored as a `char[][]`.
- Players X and O take turns marking an empty cell.
- Three of your marks in a row, column, or diagonal win.
- A full board with no winner is a draw.

Every move is one assignment, `board[row][col] = player`; the rest of the program is checks around it. The main loop:

```
public class TicTacToe {   // import java.util.Scanner;
    static Scanner scanner = new Scanner(System.in); // single Scanner

    public static void main(String[] args) {
        char[][] board = new char[3][3];   // 3x3 game board
        char player = 'X';                   // X moves first
        boolean gameOn = true;
        initBoard(board);                  // every cell ' '
        while (gameOn) {
            showBoard(board);
            makeMove(board, player);
            gameOn = isGameOn(board, player);
            player = (player == 'X') ? 'O' : 'X';   // switch turns
        }
        showBoard(board);
    }
```

`main` creates the board, lets X start, and repeats until the game ends: show the board, take a move, check for the end, pass the turn. The **conditional operator** `(player == 'X') ? 'O' : 'X'` gives O if the player is X, otherwise X.

A board made with `new char[3][3]` starts as `'\u0000'` (3.2.1), invisible and not a space, so `initBoard` fills it with `' '` before play. The `scanner` field is `static` because static methods use it; the whole program shares one `Scanner`.

```
static void initBoard(char[][] t) {
  for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
      t[i][j] = ' ';
}
static void showBoard(char[][] t) {
  System.out.println("-------------");
  for (int i = 0; i < 3; i++) {
    System.out.print("| ");
    for (int j = 0; j < 3; j++)
      System.out.print(t[i][j]+" | ");
    System.out.println("\n-------------");
  }
}
```

`initBoard` writes a space into all nine cells; `showBoard` prints the board framed by lines. In `"\n----------"`, `\n` is the newline character, so each row ends before its separator line.

```
static void makeMove(char[][] t,
                     char player) {
  int row, col;
  do {
    System.out.print("Player " + player
      + ": row and column (1-3): ");
    row = scanner.nextInt() - 1;
    col = scanner.nextInt() - 1;
  } while (row < 0 || row > 2
        || col < 0 || col > 2
        || t[row][col] != ' ');
  t[row][col] = player;
}
// the user enters 1-3, the array uses 0-2;
// invalid/occupied cell → ask again
```

`makeMove` asks until it gets a valid move, using a **do-while loop**, whose body runs once before the test — natural, since there is nothing to check before the first question. The user types 1–3 and the array uses 0–2, hence the −1. The loop repeats if the row or column is invalid or the cell is taken. The order matters: `||` evaluates left to right and stops at the first true condition, so with the index checks first, `t[row][col]` is never accessed with a bad index.

3.4.9 Checking for a win or a draw

A 3 × 3 board has 8 winning lines: 3 rows, 3 columns, 2 diagonals. A loop checks rows and columns, and a single `return` the diagonals:

```
static boolean hasWon(char[][] t, char o) {
    for (int i = 0; i < 3; i++)
        if ((t[i][0] == o && t[i][1] == o && t[i][2] == o)     // row i
         || (t[0][i] == o && t[1][i] == o && t[2][i] == o))    // column i
            return true;
    return (t[0][0] == o && t[1][1] == o && t[2][2] == o)      // main diagonal
        || (t[0][2] == o && t[1][1] == o && t[2][0] == o);     // anti-diagonal
}
static boolean isGameOn(char[][] t, char o) {
    if (hasWon(t, o)) {
        System.out.println("Congratulations, player " + o + " wins!");
        return false;
    }
    for (char[] row : t)
        for (char c : row)
            if (c == ' ') return true;          // an empty cell → continue
    System.out.println("The game is a draw.");
    return false;
}
```

For each `i`, `hasWon` tests row `i` (`t[i][0]`, `t[i][1]`, `t[i][2]`) and column `i` (`t[0][i]`, `t[1][i]`, `t[2][i]`) — the indices simply trade places, as in the transpose. Failing that, it checks the main diagonal `[0][0]`, `[1][1]`, `[2][2]` and the anti-diagonal `[0][2]`, `[1][1]`, `[2][0]`.

`isGameOn` first asks whether the player who just moved has won; if so, it congratulates them and returns `false`. Otherwise one empty cell is enough to continue, so it returns `true` at the first; with none, the game is a draw. Only the last mover needs checking, since a move can win only for the player who made it.

We trace the win check on a board where X has just moved and the rows are `X O _` / `_ X O` / `_ _ X` (`_` is empty).

Check	First failing cell	Skipped	Result
`i = 0`, row 0	`t[0][1]` = 'O'	1	false
`i = 0`, column 0	`t[1][0]` = empty	1	false
`i = 1`, row 1	`t[1][0]` = empty	2	false
`i = 1`, column 1	`t[0][1]` = 'O'	2	false
`i = 2`, row 2	`t[2][0]` = empty	2	false
`i = 2`, column 2	`t[0][2]` = empty	2	false
main diagonal	none: all three cells are 'X'	—	**true** → `return true`

Table 3.19: Trace of `hasWon(t, 'X')`. “Skipped” counts the cells never read because `&&` short-circuits.

The table shows **short-circuit** evaluation. Once one of three conditions joined by `&&` fails, the whole is false and the rest are skipped: in row 0, `t[0][1]` is 'O', so `t[0][2]` is never read; in row 1, `t[1][0]` is already empty. In the last row the left side of `||`, the main diagonal, is true, which settles the result, so the anti-diagonal is never examined and the method returns `true`.

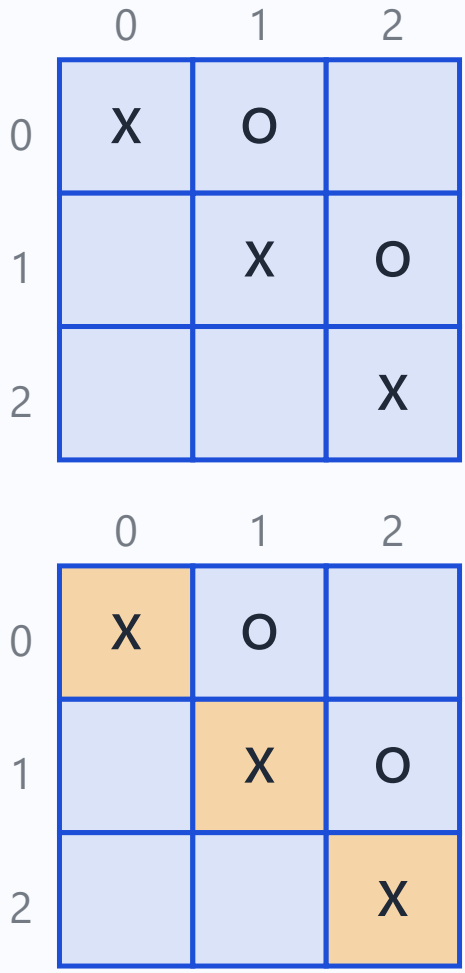


Figure 3.25: Start and end of the `hasWon(t, 'X')` trace. Left, the board: row 0 is X O empty, row 1 empty X O, row 2 empty empty X. Right, the main diagonal (`[0][0]` , `[1][1]` , `[2][2]`) highlighted: all three cells are X.

On a fixed 3 × 3 board, `hasWon` checks at most 8 lines or 24 cells, constant work. On an $n \times n$ board needing n in a line, $2n + 2$ lines of n cells would cost $O(n^2)$; checking only the last move's row, column, and diagonals (if any) takes $O(n)$.

Common mistake

Checking for a draw before a win. The last move may fill the last empty cell and complete a line at once; asking "any empty cell?" first would declare a draw although the mover has won. That is why `isGameOn` calls `hasWon` first.

3.5 The complexity of array operations

Each algorithm so far ended with a cost: *O(1)*, *O(n)*, *O(n log n)*. This section explains the notation briefly and justifies the cost of each basic array operation, so that "is this fast?" can be answered from what the operation does to the array, not from memory.

3.5.1 What array operations cost

The **complexity** of an algorithm describes how its running time and memory grow with the input size n. Seconds vary between machines, so we count basic steps — comparisons, assignments, additions — and ask how fast that count grows with n.

Definition 3.8

An algorithm's running time is *O(f(n))* if, for every sufficiently large n, its number of steps does not exceed $c \cdot f(n)$ for some constant c. This is **big-O notation**. Constant factors and lower-order terms are dropped: an algorithm taking $3n + 5$ steps is $O(n)$.

The common growth rates:

- *O(1)*, **constant**: the step count does not depend on n.
- *O(log n)*, **logarithmic**: doubling n adds one step.
- *O(n)*, **linear**: doubling n doubles the steps.
- *O(n log n)*: slightly faster than linear; the cost of good sorting algorithms.
- $O(n^2)$, **quadratic**: doubling n quadruples the steps.

For a million elements, $\log_2 n$ is about 20, n a million, n^2 a trillion. At a billion steps a second, a linear algorithm takes a thousandth of a second and a quadratic one about 17 minutes.

Time can also depend on the input itself: a linear search ends at once if the target comes first but scans everything if it is absent. Costs are therefore given for the **best case**, **average case**, and **worst case**; unless stated otherwise, a cost means the worst case, a guarantee that no input is slower. Table 3.20 summarizes array operations.

Operation	Time	Why
Access / update by index `a[i]`	*O(1)*	the address is computed directly
Search for a value (unsorted)	*O(n)*	every element may have to be examined
Search for a value (sorted, binary search)	*O(log n)*	the range halves at each step
Insert at / delete from the end (if room)	*O(1)*	no shifting
Insert / delete in the middle	*O(n)*	the following elements shift
Sorting (a good algorithm)	*O(n log n)*	e.g. merge sort

Operation	Time	Why
Copying / growing	*O(n)*	every element moves to the new array

Table 3.20: Time cost of array operations.

3.5.2 Access: O(1)

```
int[] arr = {10, 20, 30, 40, 50};
int value = arr[2];      // 30 — in one step, O(1)
```

Reading `arr[2]` involves no counting from the start: the formula of Section 3.1.2 adds index × element size to the base address — one multiplication, one addition, one memory read, whatever the array's size. Access is *O(1)*, and so is writing, `arr[2] = 35`.

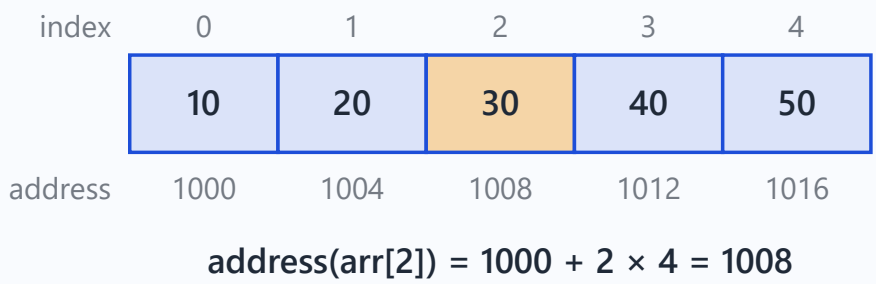


Figure 3.26: An `int` array starting at address 1000, its elements 4 bytes apart at 1000, 1004, 1008, 1012, 1016. The address of `arr[2]`, 1000 + 2 × 4 = 1008, takes one step.

Not every structure has this. A linked list scatters its elements in memory, each knowing only where the next one is, so reaching element k takes k steps from the start: *O(n)* access. When a problem indexes often, the array is the natural choice.

3.5.3 Linear search: O(n)

Access reads an element at a known index; search asks the reverse: is a value present, and where? An unsorted array gives no hint, so the only way is to check the elements one by one from the start. This is **linear search**.

```
static boolean linearSearch(int[] arr,
                            int target) {
    for (int x : arr)
        if (x == target)
            return true; // found
    return false;
}
```

The method returns `true` the moment an element matches, ignoring the rest. Only if the loop runs through the whole array is the target absent, which is why `return false` sits *outside* the loop.

We search {7, 2, 9, 5, 1, 8} for 5, then for 3.

Call	Comparisons	Count	Result
`linearSearch(arr, 5)`	7 ≠ 5, 2 ≠ 5, 9 ≠ 5, 5 == 5	4	`true`
`linearSearch(arr, 3)`	7, 2, 9, 5, 1, 8 all ≠ 3	6	`false`

Table 3.21: Linear search in {7, 2, 9, 5, 1, 8}.

5 is found at index 3 after 4 comparisons; 1 and 8 are never examined. 3 is absent, and being sure of that takes all $n = 6$ comparisons.

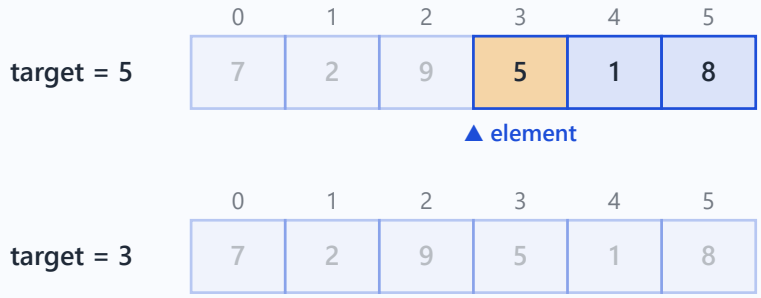


Figure 3.27: Two searches. Left, target 5: 7, 2, 9 examined (faded), 5 found at index 3, and 1 and 8 never examined. Right, target 3: all six elements examined, target not found.

- **Best case:** the target is first: 1 comparison, *O(1)*.
- **Worst case:** the target is last or absent: n comparisons, *O(n)*.
- **Average case:** a present target, equally likely anywhere, takes *(n + 1)/2* comparisons on average — still *O(n)*.

A sorted array allows binary search (Section 3.2.7), which halves the range each step and finishes in *O(log n)*. But sorting costs *O(n log n)*, more than one linear search, so it pays only when many searches will follow.

Common mistake

Putting `return false` in an `else` inside the loop: `if (x == target) return true; else return false;`. The method then checks only the first element. “Not found” can be decided only after every element has been examined.

3.5.4 Nested loops: common elements of two arrays

The most direct way to find the elements two arrays share is to compare every element of one with every element of the other:

```
static void printMatches(int[] arr1, int[] arr2) {
    for (int x : arr1)          // n times
        for (int y : arr2)      // m times
            if (x == y)
                System.out.println(
                    "Match: " + x);
}
```

The outer loop runs n times, once per element of `arr1`, and each time the inner loop visits all m elements of `arr2`: $n \cdot m$ comparisons regardless of the input, since the code never exits early.

Tracing `printMatches({3, 7, 5}, {5, 1, 3, 8})`:

x	Inner loop comparisons	Match	Total comparisons
3	3 ≠ 5, 3 ≠ 1, 3 == 3, 3 ≠ 8	3	4
7	7 ≠ 5, 7 ≠ 1, 7 ≠ 3, 7 ≠ 8	—	8
5	5 == 5, 5 ≠ 1, 5 ≠ 3, 5 ≠ 8	5	12

Table 3.22: Trace of `printMatches({3, 7, 5}, {5, 1, 3, 8})`.

It prints “Match: 3” and “Match: 5” after 3 × 4 = 12 comparisons. Figure 3.28 shows them as a grid: rows are elements of `arr1`, columns elements of `arr2`, each cell one comparison.

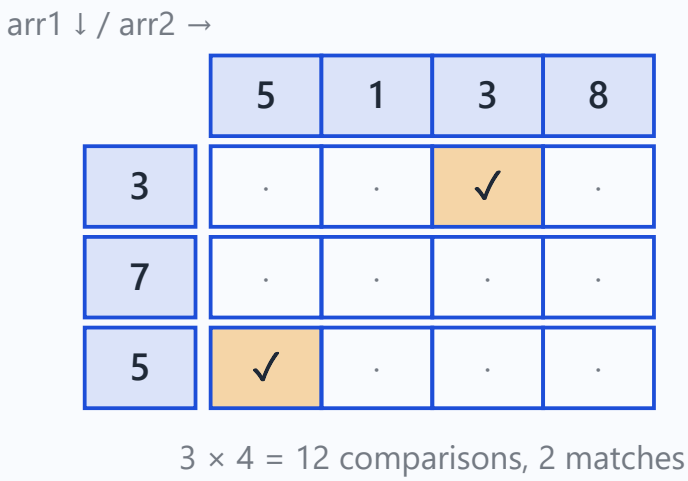


Figure 3.28: The comparison grid of the nested loops. Rows are `arr1` (3, 7, 5), columns `arr2` (5, 1, 3, 8). Each of the 3 × 4 = 12 cells is one comparison; two cells (3 with 3, 5 with 5) are matches.

For two arrays of n elements that is n^2 comparisons, $O(n^2)$: fine for small arrays, but 10 billion for two of 100,000. Two better ways:

- Sort the second array and binary-search it for each element of the first: $O(n \log n)$ plus $O(n \log n)$, so $O(n \log n)$.
- Put the second array in a hash set and look up each element of the first; at $O(1)$ per lookup on average, the total is $O(n)$ on average.

> **Common mistake**
>
> Adding the costs of nested loops. If a loop of m iterations runs *inside* one of n, the body runs $n \cdot m$ times, not $n + m$; only consecutive loops add up to $n + m$.

3.5.5 Insertion: shifting right

With a fixed size, "insertion" needs unused room. In practice an array often holds fewer elements than its capacity, the used cells packed at the front and their count kept in a separate variable (`size`), as in the dynamic array of Section 3.2.11.

To insert in the middle, every element after the insertion point shifts one cell right, and the new value goes into the freed cell:

```
static int insert(int[] a, int size,
                  int index, int value) {
  // precondition: size < a.length
  for (int k = size; k > index; k--)
    a[k] = a[k - 1];   // shift right
  a[index] = value;
  return size + 1;   // new size
}
```

The parameters are the array `a`, its used-cell count `size`, the insertion point `index`, and the `value`. The **precondition** says the array must have free room. The loop runs `k` down from `size` to `index`, copying `a[k - 1]` into `a[k]`; then the value goes into `a[index]` and the new size is returned.

We insert 4 at index 1 of `a` = {7, 2, 9, 5, 1, _} (capacity 6, five used): `insert(a, 5, 1, 4)`.

`k`	**Condition** `k > 1`	**Action**	`a`
—	—	start	[7, 2, 9, 5, 1, _]
5	true	`a[5] = a[4]`	[7, 2, 9, 5, 1, 1]
4	true	`a[4] = a[3]`	[7, 2, 9, 5, 5, 1]
3	true	`a[3] = a[2]`	[7, 2, 9, 9, 5, 1]
2	true	`a[2] = a[1]`	[7, 2, 2, 9, 5, 1]
1	false → ends	`a[1] = 4`	[7, 4, 2, 9, 5, 1]

Table 3.23: Trace of `insert(a, 5, 1, 4)`. `_` is an unused cell (value 0).

The method returns 6 after four shifts of 1, 5, 9, and 2. When the loop ends, `a[1]` and `a[2]` briefly both hold 2, until `a[1] = 4`.

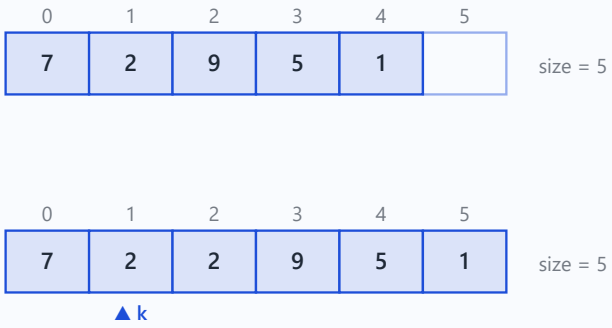




Figure 3.29: Insertion. (a) Start: capacity 6, five cells used (7, 2, 9, 5, 1), `size` = 5. (b) After four shifts: the loop has stopped at `k` = 1, and `a[1]` and `a[2]` are both 2. (c) After `a[1] = 4` : 7, 4, 2, 9, 5, 1 and `size` = 6.

The shifting must start from the right. From the left, the first step, `a[2] = a[1]` , would overwrite 9 with 2 and lose it, and the 2 would then spread rightward. From the right, each element moves before it can be overwritten: the first copy fills the empty cell, each later one the spot just vacated.

There are `size – index` shifts, so the cost depends on position:

- **At the front** (`index` = 0): *n* shifts, *O(n)*, the worst case.
- **In the middle**: about *n/2* shifts, still *O(n)*.
- **At the end** (`index` = `size`): no shifts, one assignment, *O(1)*.

A full array (`size == a.length`) must first be copied into a larger one, in *O(n)*. Doubling the capacity each time makes such copies rare enough that appending stays constant-time on average.

3.5.6 Deletion: shifting left

Deletion reverses insertion: to leave no gap, every element after the deleted position shifts one cell left.

```
static int remove(int[] a, int size, int index) {
  for (int k = index; k < size - 1; k++)
    a[k] = a[k + 1];   // shift left
  a[size - 1] = 0;    // clear the old last cell
  return size - 1;    // new size
}
```

This time the shifting starts from the **left**: the first copy overwrites the deleted value, exactly as intended, and each later copy overwrites the spot just vacated. Starting from the right would lose elements.

After the insertion above, `a` = {7, 4, 2, 9, 5, 1}. We delete the 4 at index 1: `remove(a, 6, 1)` .

`k`	**Condition** `k < 5`	**Action**	`a`
—	—	start	[7, 4, 2, 9, 5, 1]
1	true	`a[1] = a[2]`	[7, 2, 2, 9, 5, 1]
2	true	`a[2] = a[3]`	[7, 2, 9, 9, 5, 1]
3	true	`a[3] = a[4]`	[7, 2, 9, 5, 5, 1]

`k`	**Condition** `k < 5`	**Action**	`a`
4	true	`a[4] = a[5]`	[7, 2, 9, 5, 1, 1]
5	false → ends	`a[5] = 0`	[7, 2, 9, 5, 1, 0]

Table 3.24: Trace of `remove(a, 6, 1)`.

After four shifts the method returns 5 and the array is back to its state before the insertion. The last cell is left with a stale copy of 1 in what is now unused space, so `a[5] = 0` clears it. For `int`s this is optional, since `size` is 5 and the cell is never read. In an object array, though, the cell must be set to `null`; otherwise it keeps a reference to a deleted object, which the garbage collector then cannot reclaim. Objects the program no longer uses but that linger in memory this way are a **memory leak**.

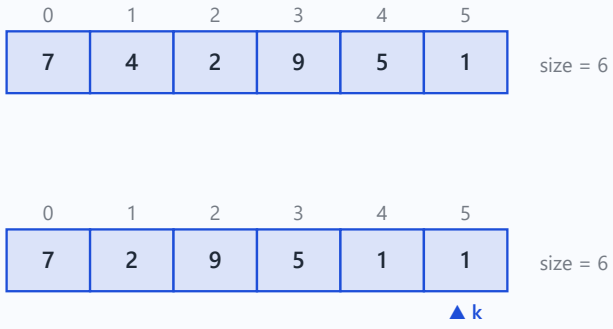


Figure 3.30: Deletion. Left, the start: 7, 4, 2, 9, 5, 1 with `size` = 6; the 4 at index 1 is to go. Right, after four shifts: 2, 9, 5, 1 have each moved left, leaving a stale 1 in the last cell, which is then cleared to 0 as `size` drops to 5.

Deletion makes `size - index - 1` shifts: $n - 1$ at the front, $O(n)$; none at the end, $O(1)$. Both insertion and deletion are dearest at the front, so arrays suit poorly a problem with frequent front insertions or deletions.

Common mistake

Shifting the wrong way. Insertion runs right to left (`k--`), deletion left to right (`k++`). Start with the cell that is safe to overwrite: the rightmost empty cell for insertion, the deleted element's cell for deletion.

3.5.7 The cost of sorting

Sorted arrays make binary search, the median, the mode, and finding duplicates easy, but sorting has its own cost, which depends on the algorithm. Table 3.25 compares five common ones.

Algorithm	Best	Average	Worst	Extra memory
Bubble sort	$O(n)$*	$O(n^2)$	$O(n^2)$	$O(1)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$**
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$

Table 3.25: Time and memory cost of sorting algorithms.

* For the version that stops after a pass with no swaps; on an already sorted array it stops after one pass.

** Quicksort's extra memory is its recursion depth: $O(\log n)$ on average, $O(n)$ in the worst case. Sorting the smaller part first keeps it within $O(\log n)$ even then.

Three observations. The three simple algorithms need no extra memory but are $O(n^2)$ on average, fit only for small arrays. Quicksort is very fast on average, but bad *pivot* choices — the pivot is the element used to split the array — can make it quadratic; always taking the first element of an already sorted array, for instance, makes every split lopsided. Merge sort is $O(n \log n)$ in every case but needs $O(n)$ extra memory.

Java's `Arrays.sort` picks by type: dual-pivot quicksort for primitive arrays (`int[]`, `double[]` …), and TimSort, a **stable** algorithm, for object arrays. A stable sort keeps equal keys in their original order: sort students by name, then by grade, and those with equal grades stay in name order. Equal primitives are indistinguishable, so they need no stability.

3.5.8 Selection sort

Selection sort splits the array into a sorted left part, initially empty, and an unsorted right part. Each pass swaps the smallest unsorted element with the first unsorted one, growing the sorted part by one. After $n - 1$ passes the last element is in place by itself.

```
int[] arr = {64, 25, 12, 22, 11};
int n = arr.length;
for (int i = 0; i < n - 1; i++) {
    int minIndex = i;
    for (int j = i + 1; j < n; j++)
        if (arr[j] < arr[minIndex])
            minIndex = j;
    // swap: by index, in place
    int temp = arr[minIndex];
    arr[minIndex] = arr[i];
    arr[i] = temp;
}
```

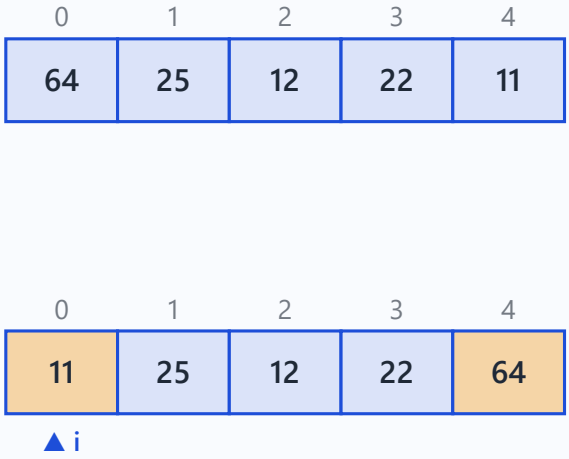
`i` marks the start of the unsorted part. `minIndex` starts at `i`, and the inner loop, from `i + 1` to the end, updates it at each smaller element. The minimum is then swapped with `arr[i]`, by index and through a temporary, as in Section 3.3.8.

Sorting {64, 25, 12, 22, 11}:

Pass (`i`)	Inner loop comparisons	`minIndex`	Swap	Array after the pass
1 (0)	25 < 64 ✓, 12 < 25 ✓, 22 < 12 ✗, 11 < 12 ✓	4	64 ↔ 11	[11, 25, 12, 22, 64]
2 (1)	12 < 25 ✓, 22 < 12 ✗, 64 < 12 ✗	2	25 ↔ 12	[11, 12, 25, 22, 64]
3 (2)	22 < 25 ✓, 64 < 22 ✗	3	25 ↔ 22	[11, 12, 22, 25, 64]
4 (3)	64 < 25 ✗	3	25 with itself	[11, 12, 22, 25, 64]

Table 3.26: Trace of selection sort on {64, 25, 12, 22, 11}.

Pass 1 swaps the minimum, 11, from the end with 64, settling index 0 for good. Pass 2 starts at index 1 and finds 12. In pass 4 the minimum is already in front (`minIndex` = `i` = 3) and is swapped with itself. The last element, 64, is in place by itself.



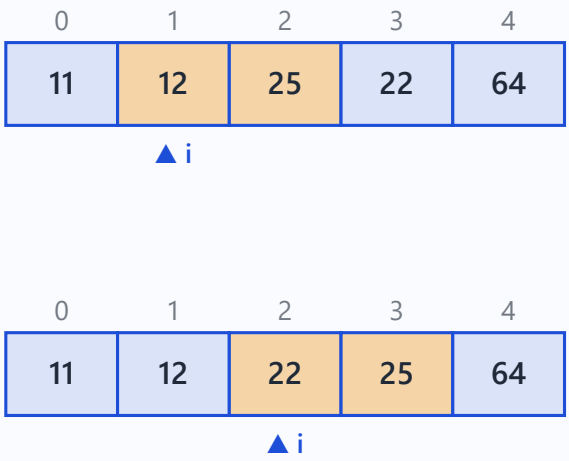


Figure 3.31: Four moments of selection sort. (a) Start: 64, 25, 12, 22, 11. (b) Swap of pass 1 (64 ↔ 11): 11 is now first. (c) Swap of pass 2 (25 ↔ 12). (d) Swap of pass 3 (25 ↔ 22): the array is 11, 12, 22, 25, 64. Highlighted cells are the elements swapped in that pass.

Theorem 3.2 (Comparisons in selection sort)

On an array of n elements, selection sort makes exactly $n(n-1)/2$ comparisons and $n-1$ swaps, whatever the input. Its time is $O(n^2)$ in every case and its extra memory $O(1)$.

Proof. In pass `i` the inner loop runs `j` from `i + 1` to `n - 1`, making $n-1-i$ comparisons. As `i` goes from 0 to $n-2$, the counts are $n-1, n-2, \ldots, 1$, which sum to $n(n-1)/2$. The loop bounds do not depend on the values in the array, so the count is the same for every input. Each pass makes exactly one swap (counting a swap with itself), for $n-1$ swaps. The algorithm uses only `i`, `j`, `minIndex`, and `temp`. □

With $n = 5$ the example made $4 + 3 + 2 + 1 = 10 = 5 \cdot 4 / 2$ comparisons, and an already sorted array would cost the same 10: selection sort never notices sorted input. But it makes at most $n-1$ swaps, an advantage when swaps are expensive, as with large records.

Common mistake

Writing a helper that swaps values: `swap(arr[minIndex], arr[i])`. Java passes arguments by value, so the method receives copies of two `int`s and changes only the copies; the array stays the same. Swap by index, either directly in the loop or with a method that takes the array and two indices, `swap(arr, i, minIndex)`, which receives the array's reference and can therefore change the array itself.

3.5.9 Insertion sort

Insertion sort works the way a card player orders a hand, sliding each new card into place among those already sorted. The array's left part is always sorted, starting as just `a[0]` (one element is always sorted); each pass inserts the next element into its proper place there.

```
static void insertionSort(int[] a) {
    for (int k = 1; k < a.length; k++) {
        int key = a[k];
        int j = k;
        while (j > 0 && a[j - 1] > key) {
            a[j] = a[j - 1];   // shift right
            j--;
        }
        a[j] = key;        // insert
    }
}
```

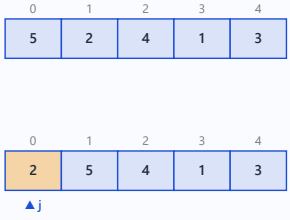
Each pass copies the next element into `key`, turning its cell into a "hole." While the element left of the hole is larger than the key, it shifts right into the hole and the hole moves left. When the left element is not larger, or the hole reaches the front (`j == 0`), the key drops in. This is the shifting of Section 3.5.5, with the insertion point found along the way.

Sorting {5, 2, 4, 1, 3}:

Pass `k`	`key`	Shifted	Stops because	Array after the pass	Shifts
1	2	5	`j == 0`	[2, 5, 4, 1, 3]	1
2	4	5	2 ≤ 4	[2, 4, 5, 1, 3]	1
3	1	5, 4, 2	`j == 0`	[1, 2, 4, 5, 3]	3
4	3	5, 4	2 ≤ 3	[1, 2, 3, 4, 5]	2

Table 3.27: Trace of insertion sort on {5, 2, 4, 1, 3}.

In pass 3 the key is 1 and the sorted part {2, 4, 5}. 5 > 1 shifts right, moving the hole to index 2; 4 > 1 moves it to 1; 2 > 1 moves it to 0. With `j` = 0 the loop stops and `a[0] = 1` is written: a key smaller than the whole sorted part shifts all of it.



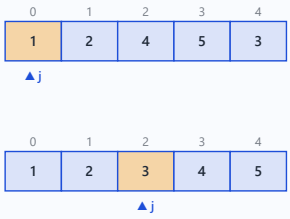


Figure 3.32: Four moments of insertion sort. (a) Start: 5, 2, 4, 1, 3. (b) End of pass 1: 2 has been placed at `j` = 0. (c) End of pass 3: 1 has been placed first; the first four elements are 1, 2, 4, 5. (d) End of pass 4: 3 has been placed at index 2 and the array is sorted.

There were 1 + 1 + 3 + 2 = 7 shifts in all. The number is not arbitrary: it equals a measure of how "unsorted" the array is.

Definition 3.9

An **inversion** in an array is a pair of indices (i, j) with $i < j$ and $a[i] > a[j]$. A sorted array has no inversions; in a reverse-sorted array of n elements every pair is an inversion, $n(n-1)/2$ in all.

The inversions of {5, 2, 4, 1, 3}, by value, are (5, 2), (5, 4), (5, 1), (5, 3), (2, 1), (4, 1), (4, 3): seven pairs, matching the seven shifts.

Theorem 3.3 (Shifts in insertion sort)

The total number of shifts insertion sort makes equals the number of inversions in the array. The algorithm therefore takes $O(n)$ time on a sorted array and $O(n^2)$ on a reverse-sorted one.

Proof. Each shift moves an element of the sorted part that is larger than the key to the key's right. That element was to the left of the key and larger than it, so the two formed an inversion. After the shift the pair is in order and stays so, because the algorithm never again changes the relative order of two elements. Moreover, every element to the left that is larger than the key is shifted, so no inversion between the key and an element on its left is missed. Thus each shift removes exactly one inversion and each inversion is removed by exactly one shift. In each pass the comparisons exceed the shifts by at most one, so the total time is $O(n + \text{number of inversions})$: $O(n)$ for a sorted array and $O(n + n(n-1)/2) = O(n^2)$ for a reverse-sorted one. □

Hence insertion sort is very fast on nearly sorted arrays: few misplaced elements mean few inversions and nearly linear time, whereas selection sort makes $n(n-1)/2$ comparisons even on sorted input. That is why library algorithms such as TimSort sort small pieces by insertion. It is also stable: `a[j - 1] > key` stops at an equal element, so equal elements keep their order.

Common mistake

Swapping the two parts of the `while` condition: `a[j - 1] > key && j > 0`. When `j` is 0, the first part tries to access `a[-1]` and throws. Since `&&` evaluates left to right, the bounds check (`j > 0`) must come first; if it is false, the array is never accessed.

3.5.10 Space complexity and memory use

Space complexity describes how the memory an algorithm or data structure needs grows with *n*, using the same notation as time complexity. An array of *n* elements takes *O(n)* space; how many bytes depends on the element type.

Array type	Per element	100 elements
`int[]`	4 bytes	400 bytes
`char[]` (UTF-16)	2 bytes	200 bytes
`double[]`, `long[]`	8 bytes	800 bytes
`Object[]` (references)	4 / 8 bytes	400 / 800 bytes + the objects

Table 3.28: Memory use by array type (data only).

Two details need explaining. A Java `char` is a UTF-16 code unit and takes 2 bytes, even for ASCII characters. Whether a reference takes 4 or 8 bytes depends on the JVM and its settings; 64-bit JVMs usually compress references to 4 bytes.

On top of the data come two more costs:

- **Overhead.** Every array object also has a header of about 16 bytes recording such things as the object's type and the array's `length`. Negligible for a large array, it adds up over many small ones.
- **Arrays of references.** In an object array such as `String[]`, only the references are contiguous. The objects they point to take separate space on the heap, each with its own header. The 400 bytes of a 100-element `String[]` are just the references; the strings themselves come on top.

Example 3.5

Let's estimate the memory of a 1000 × 1000 `int` matrix, assuming 4-byte references and 16-byte headers.

1. **Data:** 1000 · 1000 = 1,000,000 `int` s at 4 bytes each: 4,000,000 bytes, about 4 MB.
2. **Row headers:** each row is a separate array: 1000 · 16 = 16,000 bytes.
3. **Outer array:** 1000 references · 4 bytes + a 16-byte header = 4016 bytes.

The total is about 4,020,016 bytes. The overhead is under 0.5% of the total; the data decides how much memory the matrix needs.

An algorithm's space cost counts only the memory it allocates beyond its input. Here, linear search, the mean, in-place reversal, insertion, deletion, selection sort, and insertion sort use *O(1)* extra memory; the median and mode use *O(n)* for their copy, and transposition *O(m · n)* for the new matrix. Time and memory often trade off: more memory can buy speed, and less can cost it.

3.6 Common mistakes

Most array mistakes have a few sources: indices start at 0, an array is an object, and `=` and `==` work on references. Table 3.29 collects the mistakes discussed throughout these notes.

Mistake	What happens	Correct form
Off-by-one	`<=` accesses `a[a.length]` on the last iteration → exception.	`for (i = 0; i < a.length; i++)`
`length` / `length()`	A field for arrays, a method for `String`; mixing them up is a compile error.	`a.length` · `s.length()`
`=` **is not a copy**	`b = a` points to the same array; changing `b` changes `a`.	`int[] b = a.clone();`
Comparing contents with `==`	`==` compares references; `false` for two arrays with equal contents.	`Arrays.equals(a, b)`
Printing an array directly	`println(a)` prints a type and a number (`[I@…`).	`Arrays.toString(a)`
`null` **rows in a jagged array**	`m[0][0]` after `new int[3][]` → `NullPointerException`.	`m[i] = new int[k];`
Binary search on an unsorted array	`binarySearch` raises no error but may return a wrong result.	`Arrays.sort(a)` first
Column count of a 2D array	Using `m[0].length` for every row is wrong for jagged arrays.	`m[i].length`
Shallow copy of a matrix	`m.clone()` shares the rows; cell changes show up in the original.	`d[i] = m[i].clone();`
Swapping by value	`swap(a[i], a[j])` changes only copies.	`swap(a, i, j)`

Table 3.29: Common array mistakes and the correct forms.

The shared lesson: an array variable is not the array but an arrow to it. `=` copies the arrow, `==` compares arrows, `println` prints the arrow. To work on the contents, go through the cells by index or use an `Arrays` method that does.

3.7 Summary

- An **array** holds a fixed number of elements of one type in **contiguous** memory slots, indexed from 0 to `length - 1`. Element *i* is at *base + i × element size*, computed in one step, so access by index is *O(1)*.

- Arrays offer fast access, no reference overhead, and cache-friendliness; their weaknesses are fixed size, shifting on insertion and deletion (*O(n)*), and wasted capacity. Java arrays are **covariant**, so a store of the wrong type is caught only at run time, with an `ArrayStoreException`.
- A Java array is an **object** on the **heap**; the variable on the stack holds only its reference. Elements of an array created with `new` get default values (0, 0.0, `false`, `'\u0000'`, `null`). `length` is a field.
- `=` copies the reference; a real copy takes `clone()`, `Arrays.copyOf`, or `System.arraycopy` (*O(n)*). `==` compares references; compare contents with `Arrays.equals` and print with `Arrays.toString`.
- The **garbage collector** collects unreachable arrays. An out-of-bounds access throws `ArrayIndexOutOfBoundsException`; Java checks every access, C does not.
- The **mean** takes one pass, *O(n)* time and *O(1)* extra memory. The **median** and **mode** are found on a sorted copy and cost the *O(n log n)* of sorting; the copy protects the caller's array.
- **In-place reversal** moves two pointers inward from the ends and makes *⌊n/2⌋* swaps: *O(n)* time, *O(1)* extra memory. The **Caesar cipher** relies on `char` values being integers and on `% 26` for wraparound.
- A Java matrix is an **array of arrays**: `m.length` is the number of rows and `m[i].length` the number of columns in row `i`. Rows may differ in length (a **jagged array**). `m.clone()` gives a **shallow copy**; a **deep copy** copies every row as well.
- Traversing, filling, or **transposing** an $m \times n$ matrix is $O(m \cdot n)$. The body of nested loops runs as many times as the product of their iteration counts.
- **Linear search** is *O(n)* in the worst case; binary search on a sorted array is *O(log n)*. Finding the common elements of two arrays with nested loops is $O(n \cdot m)$.
- **Insertion** shifts right to left, **deletion** left to right; both are *O(n)* in the worst case and *O(1)* at the end.
- **Selection sort** makes *n(n − 1)/2* comparisons on every input. **Insertion sort** makes as many shifts as there are inversions: *O(n)* on a sorted array, $O(n^2)$ in the worst case. Both use *O(1)* extra memory.

Operation	Time	Extra memory
Access / update `a[i]`	*O(1)*	*O(1)*
Copying (`clone` , `copyOf`)	*O(n)*	*O(n)*
Mean	*O(n)*	*O(1)*
Median, mode	*O(n log n)*	*O(n)*
In-place reversal	*O(n)*	*O(1)*
Linear search	*O(n)*	*O(1)*
Binary search (sorted array)	*O(log n)*	*O(1)*
Insertion / deletion in the middle	*O(n)*	*O(1)*
Selection sort	$O(n^2)$	*O(1)*

Operation	Time	Extra memory
Insertion sort	$O(n)$ … $O(n^2)$	$O(1)$
Transpose ($m \times n$)	$O(m \cdot n)$	$O(m \cdot n)$

Table 3.30: Costs of the operations in this chapter.

3.8 Key terms

Each term links to the course glossary, which gives its Turkish equivalent and its use in other chapters.

array · element · index · initializer · capacity · static array · dynamic array · reference · covariance · heap memory · stack frame · two-dimensional array · multidimensional array · matrix · jagged array · shallow copy · deep copy · transpose · median · mode · in-place · swap · linear search · binary search · bubble sort · selection sort · insertion sort · quicksort · merge sort · stable (sort) · time complexity · space complexity

3.9 Test Yourself

1. After `int[] a = {4, 7, 1}; int[] b = a; a = new int[] {0}; b[1] = 9;`, which arrays do `a` and `b` point to?

   Answer

   `a` = [0], `b` = [4, 9, 1]. After `b = a` both point to the first array. Assigning a new array to `a` changes only `a`'s reference, so `b` still points to the first array, which `b[1] = 9` then changes. The new array is unaffected.

2. What do `Arrays.toString(new double[3])`, `Arrays.toString(new boolean[2])`, and `Arrays.toString(new String[2])` print?

   Answer

   `[0.0, 0.0, 0.0]`, `[false, false]`, and `[null, null]`. Elements of an array created with `new` get their type's default value, and the default for `String` is `null`, not the empty string.

3. What are the mean and median of {10, 2, 7, 3, 100, 1}? Which better describes the "middle" of the

data?

Answer

The sum is 123, so the mean is 123 / 6 = 20.5. Sorted, the array is {1, 2, 3, 7, 10, 100}; the size is even, so the median is the mean of 3 and 7 at indices 2 and 3: 5.0. Five of the six values are 10 or less; the 100 has pulled the mean up. The median ignores the extreme value and describes the middle better.

4. What does `mode({8, 3, 8, 5, 3, 8})` return? How many times is the record updated during the loop?

Answer

The sorted copy is {3, 3, 5, 8, 8, 8}. At `i = 1`, 3 is seen a second time and the record becomes 3 (twice). At `i = 2` and `i = 3` new values begin and the counter is 1. At `i = 4`, 8 has been seen twice, but `2 > 2` is false. At `i = 5`, 8 is seen a third time and the record becomes 8 (three times). The method returns 8; the record changes twice in the loop.

5. What does `encrypt("JAVA", 5)` return? With what *k* must the same method be called to decrypt?

Answer

J: 9 + 5 = 14 → O. A: 0 + 5 = 5 → F. V: 21 + 5 = 26, 26 % 26 = 0 → A. A → F. The result is `"OFAF"`. To decrypt, use *k = 26 − 5 = 21*: `encrypt("OFAF", 21)` → `"JAVA"`.

6. If the Pascal's triangle code runs with `n = 6`, what is row `p[5]`? How many cells does the triangle use?

Answer

The rows are 1; 1 1; 1 2 1; 1 3 3 1; 1 4 6 4 1; 1 5 10 10 5 1, so `p[5]` = [1, 5, 10, 10, 5, 1]; for example, `p[5][2] = p[4][1] + p[4][2] = 4 + 6 = 10`. It uses 1 + 2 + … + 6 = 21 cells, where a 6 × 6 square matrix would use 36.

7. A 3 × 3 × 3 array holds `x * 9 + y * 3 + z + 1`. Which cell holds 20?

Answer

Since *20 − 1 = 19 = 2 · 9 + 0 · 3 + 1*, `x` = 2, `y` = 0, `z` = 1: `array3D[2][0][1]`, the second cell of the first row of the third layer.

8. With `a` = {5, 6, 8, 9, 0} and `size` = 4, `insert(a, 4, 0, 7)` is called. How many shifts are made,

and what is the array? What happens if `remove(a, 5, 4)` is called next?

Answer

Inserting at the front shifts all four elements: 4 shifts, for `k` = 4, 3, 2, 1. The array becomes {7, 5, 6, 8, 9} and the method returns 5. `remove(a, 5, 4)` deletes the last element: the loop condition `4 < 4` is false from the start, so nothing shifts. `a[4] = 0` is written, the array becomes {7, 5, 6, 8, 0}, and the method returns 4. Inserting at the front is *O(n)*; deleting at the end is *O(1)*.

9. {4, 3, 2, 1} is sorted first with selection sort, then (starting from the same array) with insertion sort. How many comparisons does selection sort make, and how many shifts does insertion sort make? What are the answers for {1, 2, 3, 4}?

Answer

Selection sort makes $n(n-1)/2 = 6$ comparisons on every input, 6 for both arrays. Insertion sort makes as many shifts as there are inversions. In {4, 3, 2, 1} every pair is an inversion: 6 shifts (1 + 2 + 3). {1, 2, 3, 4} has no inversions: 0 shifts. Then each pass ends after one comparison, for 3 comparisons in total.

10. How many bytes is the data of a matrix created with `new int[1000][1000]`? In the worst case, how many comparisons does searching it for a value cell by cell take?

Answer

The data is 1,000,000 × 4 = 4,000,000 bytes (about 4 MB), plus the headers of the 1000 row arrays and the outer array and the 1000 references in the outer array. A search that visits every cell makes 1,000,000 comparisons when the value is absent: $O(r \cdot c)$ for an $r \times c$ matrix.

3.10 Further reading

- M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, *Data Structures and Algorithms in Java* (6th ed., 2014), Section 3.1: covers topics from these notes — sorting arrays, `java.util.Arrays`, simple encryption with character arrays, and board games on two-dimensional arrays — in Java code.
- R. Sedgewick and K. Wayne, *Algorithms* (4th ed., 2011), Sections 1.1 and 2.1: using arrays in Java, the distinction between reference and copy, and a comparative analysis of selection and insertion sort, with clear examples.
- J. Gosling et al., *The Java Language Specification, Java SE 17 Edition*, Chapter 10: the official definition of array declaration, initialization, default values, the shallow-copy behavior of `clone`, and `ArrayStoreException`.

3.11 References

1. M. T. Goodrich, R. Tamassia, M. H. Goldwasser, *Data Structures and Algorithms in Java*, 6th ed., Wiley, 2014.
2. J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, D. Smith, G. Bierman, *The Java Language Specification, Java SE 17 Edition*, Oracle, 2021.
3. R. Lafore, *Data Structures and Algorithms in Java*, 2nd ed., Sams, 2002.
4. Oracle, *Java SE 17 API Specification*: documentation for `java.util.Arrays` and `java.lang.System`.
5. R. Sedgewick, K. Wayne, *Algorithms*, 4th ed., Addison-Wesley, 2011.

PART I · CHAPTER 4

Recursion

Recursion solves a problem by reducing it to a smaller instance of itself; in Java, a method calls itself on a smaller input. How can a method use itself before it is finished? On the smallest input it answers directly; on larger inputs it takes the smaller problem's answer and builds its own in one step. We start from this idea and trace how the call stack runs the calls. Then come the kinds of recursion: linear (array sum, reversal, fast exponentiation), binary (Fibonacci, binary search), the Towers of Hanoi, and backtracking for subsets and permutations. The closing sections treat cost — recurrence relations, call trees, space, tail recursion, converting recursion to a loop — and recursion on linked lists leads into the structures ahead.

Code for this chapter: `github.com/sercankulcu/data-structures-java/Bolum04/src`

LEARNING OUTCOMES

Recursive design
Define a problem by a base case and a recursive step, and turn that definition into a correct Java method.

The call stack
Trace a recursive method frame by frame to find its output and return value, and say how many frames the stack reaches at most.

Kinds of recursion
Tell linear, binary, and multiple recursion and backtracking apart, with examples.

Cost
Write a recurrence relation and use the call tree to find time and space cost; turn a recursive method into a loop when needed.

4.1 What is recursion?

Methods call other methods: `main` calls `System.out.println`, a sort calls a helper that swaps two elements. In recursion, the method called is the caller itself. Why doesn't it loop forever, how do we design one, and how does it run line by line?

4.1.1 An everyday example

You are in a long line at a box office and want to know your position. You can't walk to the front without losing your place, and you can see only the person directly ahead.

So you ask them, “What's your position?” If they don't know, they ask the person ahead, and the question moves forward. The person at the front has nobody ahead and simply says “1.” The answer then travels back, each person adding 1 to what they hear (“you're 1st, so I'm 2nd”), until it reaches you.

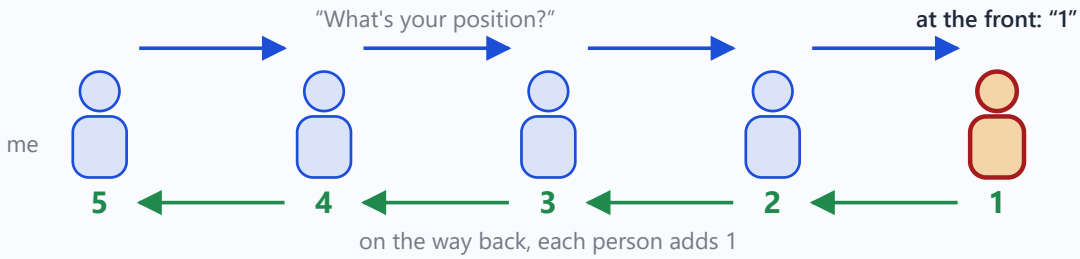


Figure 4.1: Finding your place in line. Blue arrows carry the question forward; the person at the front (orange) answers “1” without asking anyone. Green arrows carry the answer back, each person adding 1: 1, 2, 3, 4, 5. The first asker (“me”) is 5th.

The process in Figure 4.1 has three parts:

- **Same question, shorter line.** Everyone follows the same rule, each time for a line one person shorter.
- **A stopping point.** The person at the front answers without passing the question on; without them, it would travel forever.
- **The way back.** Each person adds their share, 1, as the answer returns. The real computation happens here.

A recursive method works the same way: it calls itself on a smaller input, answers the smallest input directly, and builds its answer from the smaller one on the way back. Each person is one call, and just as each person remembers something of their own, each call has its own variables.

4.1.2 Base case and recursive step

The classic example is the factorial: $n!$ is the product of the integers from 1 to n, so $4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$. Since $4!$ contains $3 \cdot 2 \cdot 1 = 3!$, we have $4! = 4 \cdot 3!$, and likewise for every $n > 1$. The factorial can thus be defined in terms of itself:

> $n! = 1$ (if $n \le 1$) $\cdot$ $n! = n \cdot (n - 1)!$ (if $n > 1$)

The first line answers the smallest inputs outright; the second builds a large answer from a smaller one. For *4!* we apply the second line three times, then the first: *4! = 4 · 3! = 4 · 3 · 2! = 4 · 3 · 2 · 1! = 4 · 3 · 2 · 1 = 24*.

Definition 4.1

Recursion is a method calling itself on a smaller input. A recursive method has two parts:

- **Base case:** inputs solved directly, without a recursive call; the recursion stops here.
- **Recursive step:** shrinks the problem toward the base case and builds the larger answer from the smaller one.

The definition translates into Java almost word for word:

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

// in main:
System.out.println(factorial(4));
```

Line 2 catches the base case and returns 1. The last `return` is the recursive step: `factorial(n - 1)` runs the same method on a smaller input, and its result is multiplied by `n`. The call in `main` prints 24.

If every call makes another call, the chain must break somewhere, and the base case is where. Without it the calls never end; each takes some memory, memory runs out, and Java stops the program with a `StackOverflowError` (Section 4.2.3).

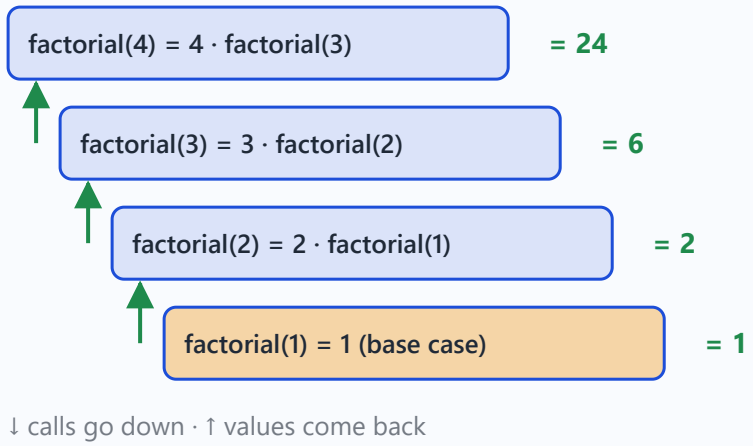


Figure 4.2: Unfolding `factorial(4)`. Calls go down, each on an input one smaller, until `factorial(1)`, the base case, returns 1 (orange). Values come back up the green arrows: 1, 2, 6, 24.

Figure 4.2 shows recursion's two directions. Going down, nothing is multiplied; each call just remembers its `n` and waits. Coming up, the multiplications happen: *2 · 1 = 2*, *3 · 2 = 6*, *4 · 6 = 24*.

At the deepest point four calls of `factorial` sit in memory, each with its own `n`: 4, 3, 2, 1. While `factorial(1)` runs with `n = 1`, the `n` of `factorial(4)` is still 4, ready for its multiplication. Each call gets its own area of memory, a **frame** (Section 4.2).

Common mistake

Thinking of a recursive method as a loop with shared variables leads to reasoning like "once `factorial(1)` is called, `n` is 1 on the `4 * ...` line too." That is wrong: every call gets fresh parameters and locals, and changing one call's variable leaves the others alone.

4.1.3 Thinking recursively

Picturing the whole chain of calls is exhausting and usually unnecessary. Ask three questions instead:

1. **Base case:** What is the smallest input, and its answer?
2. **Reduction:** How does the problem reduce to a smaller one of the same kind? Typically to *n − 1*, *n / 2*, or index *i + 1*.
3. **Combination:** How is the larger answer built from the smaller one?

For the third question we *assume* the smaller problem is solved correctly. Writing `factorial(n)`, we don't ask how `factorial(n - 1)` works; we trust it to return *(n − 1)!* and think only about the last step: multiply by *n*.

This is no trick; it is **mathematical induction**. To prove a statement for every natural number, show it for the smallest one, and show that if it holds for $n - 1$ it holds for n. Then it holds for 1, hence 2, hence 3, without a break. Likewise, if the base case is right and each step builds a correct answer from a correct smaller one, the method is correct for every input.

One more condition: every recursive call must move the input *closer* to the base case. If the input doesn't shrink, or skips past the base case, the chain never reaches its start.

Take a simple method: `countdown(n)` prints n down to 1, one per line, then "Done".

Example 4.1

1. **Base case.** If $n = 0$, print "Done" and stop.
2. **Reduction.** Counting down from n means printing n, then counting down from $n - 1$: `countdown(n - 1)`.
3. **Combination.** Nothing is returned; the order matters. Print n first, then solve the smaller problem.

Each call decreases n by 1, so a positive n reaches 0 in finitely many steps.

The three answers turn directly into code:

```
void countdown(int n) {
  if (n == 0) {     // base case
    System.out.println("Done");
    return;
  }
  System.out.println(n);
  countdown(n - 1); // recursive step
}
countdown(3);       // in main
```

Lines 2–5 are the base case: print "Done" and leave. The `return` matters; without it, execution would fall through to line 6, print 0, and call `countdown(-1)`. Line 6 prints n; line 7, the recursive step, solves the smaller problem.

Common mistake

A base case designed only for "normal" inputs can fail. Given a negative number, `countdown` counts −1, −2, … and never reaches 0. If negative inputs are possible, write the base case as `n <= 0`.

4.1.4 Code trace: countdown(3)

The surest way to understand recursion is to trace a small input line by line. Here `main` calls `countdown(3)`; at each step we note the running call, the waiting ones, and the output.

Step	Event	Calls	Output
1	`main` calls `countdown(3)`	main	
2	*n = 3 ≠ 0*: print 3, call `countdown(2)`	main, cd(3)	3
3	*n = 2 ≠ 0*: print 2, call `countdown(1)`	main, cd(3), cd(2)	3 2
4	*n = 1 ≠ 0*: print 1, call `countdown(0)`	main, cd(3), cd(2), cd(1)	3 2 1
5	*n = 0*: base case, print "Done"	main, cd(3), cd(2), cd(1), cd(0)	3 2 1 Done
6	`countdown(0)` ends with `return`	main, cd(3), cd(2), cd(1)	3 2 1 Done
7	`countdown(1)` returns from line 7 and ends	main, cd(3), cd(2)	3 2 1 Done
8	`countdown(2)` returns and ends	main, cd(3)	3 2 1 Done
9	`countdown(3)` returns; back in `main`	main	3 2 1 Done

Table 4.1: Tracing `countdown(3)`. The Calls column lists the calls in memory, outermost first; the rightmost is running.

Here `cd` abbreviates `countdown`. The Calls column grows in steps 2–5 and shrinks in steps 6–9; at its longest, 5 calls are in memory, counting `main`. In total `countdown` is called 4 times: *n = 3, 2, 1, 0*.

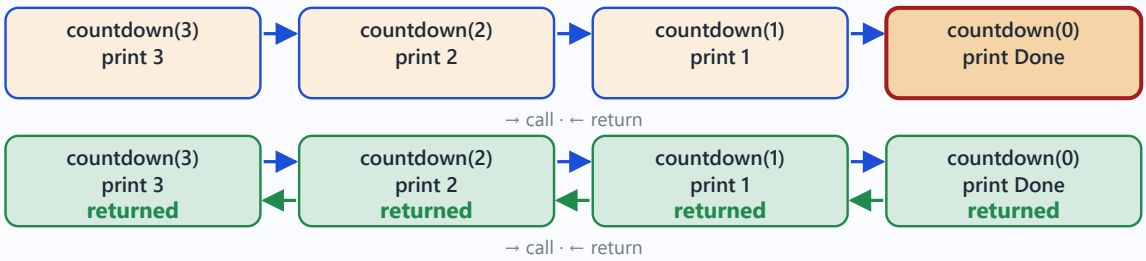


Figure 4.3: The call chain of `countdown(3)`. Top: the deepest moment (Table 4.1, step 5); three calls wait while `countdown(0)`, outlined in red, prints "Done." Bottom: all four calls have returned (green). Blue arrows are calls, green arrows returns.

Nothing is printed on the way back (steps 6–9): each call returns from line 7 at the end of the method with nothing left to do. All printing happens on the way down, because the print comes *before* the call. Swap the two lines and the output changes.

4.1.5 Work before or after the call?

Whether work comes before or after the recursive call decides the result. These two methods differ only in where the print goes.

```
void countdown2(int n) {
  if (n == 0) return;
  System.out.print(n + " ");
  countdown2(n - 1);   // call after
}
```

```
void countUp(int n) {
  if (n == 0) return;
  countUp(n - 1);      // call first
  System.out.print(n + " ");
}
```

`countdown2(3)` prints `3 2 1`: each call prints its number before moving on, so printing happens as the calls go *down*.

`countUp(3)` prints `1 2 3`. Each call first makes the recursive call and waits: `countUp(3)` waits for `countUp(2)`, which waits for `countUp(1)`, which calls `countUp(0)`, a base case that returns at once. So `countUp(1)` prints first, then `countUp(2)` resumes and prints 2, then `countUp(3)` prints 3. Printing happens as the calls come *back*.

Example 4.2

Tracing `countUp(3)`:

1. `countUp(3)`: $n \neq 0$, call `countUp(2)`. Printing 3 is postponed.
2. `countUp(2)`: call `countUp(1)`. Printing 2 is postponed.
3. `countUp(1)`: call `countUp(0)`. Printing 1 is postponed.
4. `countUp(0)`: base case, returns at once.
5. `countUp(1)` resumes: prints `1` and ends.
6. `countUp(2)` resumes: prints `2` and ends.
7. `countUp(3)` resumes: prints `3` and ends.

Output: `1 2 3`. At the deepest point memory holds `main` and four `countUp` calls.

The first call to finish is the last to start, so work after the call runs in reverse order of the calls. That pending work waits in memory with its call until its turn. This is what lets us print a singly linked list backward (4.8.3) and move the second half of the disks in the Towers of Hanoi (4.5.3).

Common mistake

It is a classic slip to expect `3 2 1` from `countUp(3)`. The calls start in that order, but since the print follows the call, none can print until its call returns. Asked for a recursive method's output, first check whether the print precedes or follows the call.

4.2 The call stack

Several calls of one method can sit in memory at once, each with its own variables. Where does Java keep them, and in what order? Once that is clear, both the output and the memory cost of a recursive method are easy to find.

4.2.1 Frames and the call stack

Each method call gets a small area of memory holding its parameters, its local variables, and the place to resume when it finishes, the **return address**. When `factorial(3)` calls `factorial(2)`, the return address is "where `n * …` is computed."

Definition 4.2

A **stack frame** (or frame) is the record of one method call on the call stack: its parameters, its local variables, and where to resume on return. Frames live in a region of memory called the **call stack**. A call pushes a new frame on top; when the method finishes, its frame is popped.

The call stack follows the rule of the stack data structure: last in, first out, or **LIFO**. Method calls behave this way by nature: if `main` calls `f` and `f` calls `g`, then `g` finishes first, then `f`, then `main`.

Three properties of the call stack explain recursion:

- **Only the top frame runs.** Each frame below waits for the call just above it.
- **One method can have several frames on the stack.** That is recursion, and each frame has its own `n`.
- **A finished call's frame is removed.** Its return value goes to the caller, now on top, which resumes at the return address.

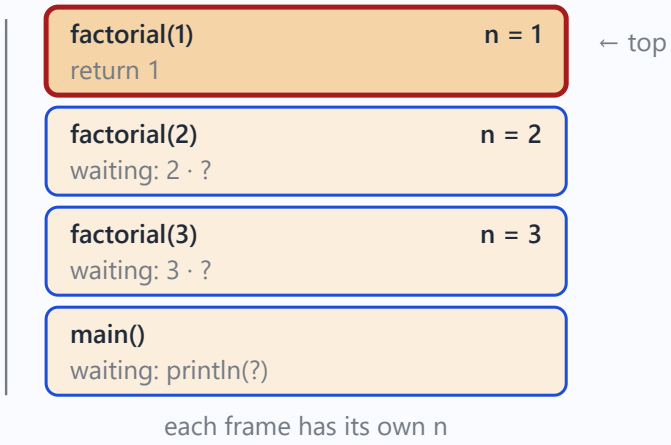


Figure 4.4: The call stack at the deepest point of `factorial(3)`: `main` at the bottom, then `factorial(3)`, `factorial(2)`, and `factorial(1)`. Only the top frame (red outline) runs, about to return 1. The others wait to finish "2 · ?", "3 · ?", and the print. Each frame has its own `n`.

Figure 4.4 also shows recursion's memory cost. Four frames are on the stack at once; for `factorial(n)` there are *n + 1*, counting `main`. Each frame has a fixed size, so memory is *O(n)*.

In general, a recursive method's memory cost depends not on the total number of calls but on the most frames on the stack *at once* — the length of the deepest call chain. In binary recursion the two numbers differ enormously (Section 4.6.5).

Common mistake

Counting stack depth, it is easy to forget `main` or the base-case call. At the deepest point of `factorial(4)` the stack holds `main` and the frames of `factorial(4)` through `factorial(1)`: 4 + 1 = 5 frames. Check whether a question counts `main`.

4.2.2 Code trace: factorial(4)

Now we trace `factorial(4)` (Section 4.1.2), watching the call stack at each step:

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

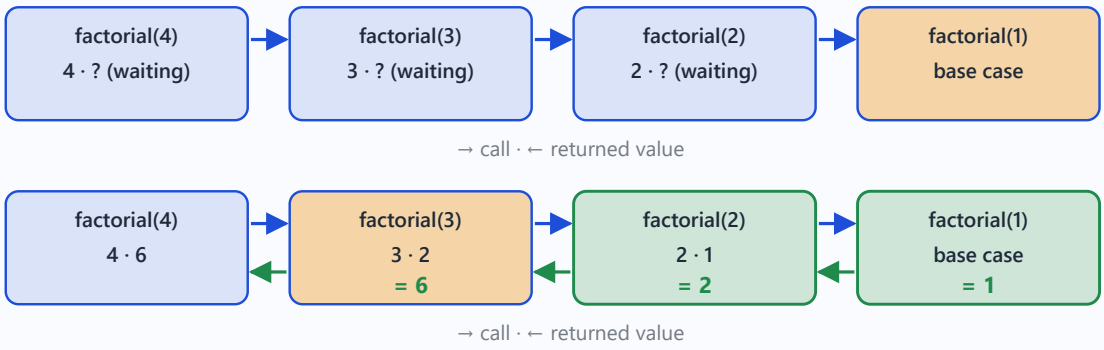
// in main:
System.out.println(factorial(4));
```

Two lines matter in each call. Line 2 separates out the base case. Line 4 calls `factorial(n - 1)`, multiplies the result by `n`, and returns it; until the call returns, the frame holds the multiplication as unfinished work.

Step	Event	Call stack (bottom to top)	Returned
1	`main` calls `factorial(4)`	main	
2	New frame: *n = 4*. Is *4 ≤ 1*? No	main, f(4)	
3	`4 * factorial(3)` : waits for `factorial(3)`	main, f(4)	
4	New frame: *n = 3*. Is *3 ≤ 1*? No	main, f(4), f(3)	
5	`3 * factorial(2)` : this frame waits	main, f(4), f(3)	
6	New frame: *n = 2*. Is *2 ≤ 1*? No	main, f(4), f(3), f(2)	
7	`2 * factorial(1)` : this frame waits	main, f(4), f(3), f(2)	
8	New frame: *n = 1*. Is *1 ≤ 1*? Yes	main, f(4), f(3), f(2), f(1)	
9	Base case: `return 1`	main, f(4), f(3), f(2), f(1)	1
10	`factorial(1)` is popped; *2 · 1 = 2*, `return 2`	main, f(4), f(3), f(2)	2
11	`factorial(2)` is popped; *3 · 2 = 6*, `return 6`	main, f(4), f(3)	6
12	`factorial(3)` is popped; *4 · 6 = 24*, `return 24`	main, f(4)	24
13	`factorial(4)` is popped; `main` prints 24	main	

Table 4.2: Tracing `factorial(4)`. The stack is written bottom to top; `f` is `factorial`.

Steps 2–8 are the **descent**: each pushes a frame, and nothing is multiplied. Step 9 reaches the base case and produces the first value, 1. Steps 10–13 are the **return**: each pops a frame, and the frame below completes its multiplication. Emptying the stack this way is called **unwinding**.



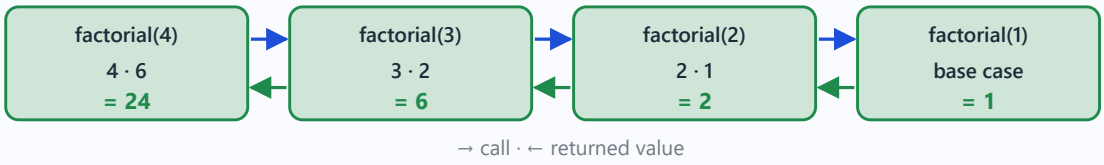


Figure 4.5: Three moments in tracing `factorial(4)`. Top: the deepest moment (Table 4.2, step 8), three calls waiting (blue), `factorial(1)` in the base case (orange). Middle: step 11; `factorial(1)` and `factorial(2)` have returned (green), and `factorial(3)` computes *3 · 2 = 6*. Bottom: `factorial(4)` returns *4 · 6 = 24*.

The cost is easy to read off: 4 calls, each doing constant work (a comparison, at most one multiplication, a return), so time is *O(n)*. At its deepest the stack holds 4 frames plus `main`, so memory is *O(n)* too. A loop needs only *O(1)* memory; the recursive version pays for short code that mirrors the definition.

Common mistake

Mind the limits of `int`, which holds at most $2^{31} - 1 \approx 2.1$ billion. *12! = 479,001,600* fits; *13! = 6,227,020,800* doesn't, and the result is silently wrong. Use `long` (up to *20!*) or `BigInteger`. This overflow is unrelated to recursion; don't confuse it with stack overflow.

4.2.3 Stack overflow

The call stack is finite. The JVM (Java Virtual Machine) gives each thread a fixed amount of stack memory, which depends on the JVM and operating system and can be set with the `-Xss` option. Every frame uses some of it, so with default settings recursion depth is usually limited to a few thousand to a few tens of thousands of calls.

The following method adds up the numbers from 1 to *n*:

```
long sumTo(int n) {        // 1 + 2 + … + n
  if (n == 0) return 0;
  return n + sumTo(n - 1);
}
// in main:
System.out.println(sumTo(10_000));   // 50005000
System.out.println(sumTo(1_000_000)); // error
```

`sumTo(10_000)` prints 50005000; its 10,001 frames fit in the default stack. `sumTo(1_000_000)` fills the stack before reaching the base case — there is no room for a million frames — and Java stops with:

```
Exception in thread "main"
  java.lang.StackOverflowError
    at Example.sumTo(Example.java:3)
    at Example.sumTo(Example.java:3)
    …
```

The lines below the message, a dump of the call stack at the failure, form a **stack trace**. One line, `Example.sumTo(Example.java:3)`, repeated thousands of times points straight at recursion.

For scale: a method that did nothing but call itself reached a depth of about 21,800 with default settings, and `sumTo`, with a slightly larger frame, less. The exact value varies by machine and doesn't matter. What matters is that a limit *exists* and its order of magnitude: tens of thousands is usually fine; millions almost always overflow.

The sum needs no recursion. A loop takes *O(n)* time and *O(1)* memory, and the formula *1 + 2 + … + n = n(n + 1)/2* takes *O(1)* time. For *n = 1,000,000* that is 500,000,500,000, too big for an `int` — hence the `long`.

Common mistake

Don't catch `StackOverflowError` with `try` – `catch` and carry on. It is an `Error`, not an `Exception`: the JVM is reporting a serious resource problem. Reduce the recursion depth (by halving the problem at each step, say) or convert the method to a loop.

4.2.4 Forgetting the base case

Stack overflow usually comes not from a huge input but from a faulty base case. Three typical faults give the same result for different reasons.

1. No base case.

```
int f(int n) {
  return n * f(n - 1);
}
```

Every call calls again. For `f(3)` the inputs run 3, 2, 1, 0, −1, …; nothing returns, and the stack fills.

2. The base case can't be reached.

```
if (n == 1) return 1;
return n * f(n - 1);
```

The base case catches only *n = 1*. For `f(0)` the inputs run 0, −1, −2, … and never reach 1. The test `n <= 1` fixes this: `f(0)` returns 1 at once. The same happens when the input drops by 2: `n * f(n - 2)` with base case `n == 1` stops for odd numbers but runs 2, 0, −2, … for even ones.

3. The problem doesn't shrink.

```
if (n <= 1) return 1;
return n * f(n);
```

The base case is right, but the step doesn't shrink the input: `f(n)` calls `f(n)`. `f(1)` works; `f(2)` calls `f(2)` forever, never nearer the base case.

Example 4.3

This method should add the even numbers up to *n*: `int g(int n) { if (n == 0) return 0; return n + g(n - 2); }`. What do `g(6)` and `g(5)` do?

1. `g(6)`: *6 ≠ 0*, `6 + g(4)`.
2. `g(4)`: `4 + g(2)`; `g(2)`: `2 + g(0)`.
3. `g(0)`: base case, 0. Coming back: *2 + 0 = 2*, *4 + 2 = 6*, *6 + 6 = 12*. `g(6) = 12`, in 4 calls.
4. `g(5)`: `5 + g(3)`; `g(3)`: `3 + g(1)`; `g(1)`: `1 + g(-1)`; … The input stays odd, never 0: `StackOverflowError`.

With base case `n <= 0`, `g(5)` stops at `g(-1)` and returns *5 + 3 + 1 + 0 = 9*. The method now always stops, but for odd *n* it sums the odd numbers; depending on the intent, require an even input or step to *n − 1* first.

The rule: every possible input, including 0 and negatives, must reach a base case in finitely many steps. After writing a recursive method, ask which inputs end directly in a base case, and whether every other input moves toward one with each call. Unless both answers are “yes,” some inputs overflow.

Common mistake

`f(5)` working doesn't mean the method works for every input. Also test 0, 1, a negative number, and, if parity matters, one odd and one even input.

4.3 Linear recursion

In every example so far, each call made at most one recursive call, so the calls form a straight, unbranched chain. This section covers three such problems — summing an array, reversing an array, computing powers — and the last shows how shortening the chain changes the cost.

4.3.1 Definition and summing an array

> **Definition 4.3**
>
> In **linear recursion** each call makes at most one recursive call. A problem of size n reduces to one problem of size $n - 1$ (or $n / 2$), and the calls form an unbranched chain.

`factorial`, `countdown`, and `sumTo` are linear, as are array sum and reversal, powers, and the length of a linked list.

To sum an integer array recursively, we first need a form that can shrink: instead of "the sum of the array," take "the sum of the first n elements," so n can be reduced. Then the three questions of Section 4.1.3 apply:

- **Base case:** for $n = 0$ the part is empty and sums to 0.
- **Reduction:** use the sum of the first $n - 1$ elements.
- **Combination:** add the last element, `a[n - 1]`.

Indices start at 0, so the first n elements are `a[0]`, …, `a[n - 1]`, and the last is `a[n - 1]`, not `a[n]`:

```
int sum(int[] a, int n) {      // sum of the first n elements
  if (n == 0)
    return 0;                   // base case
  return sum(a, n - 1) + a[n - 1];
}
```

The second parameter says which part we are working on. The array is never changed or copied: Java passes arrays by reference, so every frame's `a` refers to the same array, and each call carries just a reference and an integer. The first call is `sum(a, a.length)`.

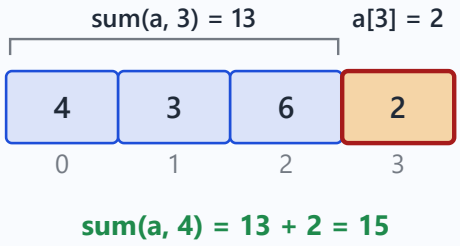


Figure 4.6: One step for `a = {4, 3, 6, 2}`: the first three elements give `sum(a, 3) = 13`, and adding `a[3] = 2` gives `sum(a, 4) = 15`.

Figure 4.6 is recursive thinking in a nutshell: to write `sum(a, 4)` you needn't know how `sum(a, 3)` got 13; you trust it and add the last element.

4.3.2 Code trace: sum(a, 4)

```
int sum(int[] a, int n) {      // sum of the first n elements
  if (n == 0)
    return 0;                   // base case
  return sum(a, n - 1) + a[n - 1];
}
// in main:
int[] a = {4, 3, 6, 2};
System.out.println(sum(a, 4));
```

Each call first asks for the sum of a smaller part and leaves its own element for the way back.

Step	Call	Event	Returned
1	`sum(a, 4)`	$n \neq 0$: first `sum(a, 3)`; `a[3] = 2` waits	
2	`sum(a, 3)`	first `sum(a, 2)`; `a[2] = 6` waits	
3	`sum(a, 2)`	first `sum(a, 1)`; `a[1] = 3` waits	
4	`sum(a, 1)`	first `sum(a, 0)`; `a[0] = 4` waits	
5	`sum(a, 0)`	base case: empty part	0
6	`sum(a, 1)`	*0 + a[0] = 0 + 4*	4
7	`sum(a, 2)`	*4 + a[1] = 4 + 3*	7
8	`sum(a, 3)`	*7 + a[2] = 7 + 6*	13
9	`sum(a, 4)`	*13 + a[3] = 13 + 2*	15

Table 4.3: Tracing `sum(a, 4)` with `a = {4, 3, 6, 2}`.

The stack is deepest at step 5: `main` and 5 `sum` frames. The additions happen on the way back, from `a[0]` to `a[3]`, because the addition follows the call, as in `countUp` (4.1.5).

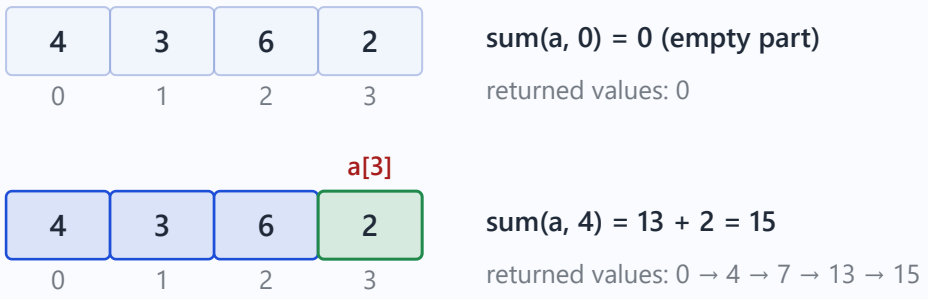


Figure 4.7: Two moments in tracing `sum(a, 4)`. Top: the base case (Table 4.3, step 5); the empty part is faded, and `sum(a, 0)` returns 0. Bottom: the returned values 0, 4, 7, 13, 15, with `a[3] = 2` added last.

Cost: *n* elements take *n + 1* calls, each doing a comparison, an addition, and an array access, so time is *O(n)*; the stack reaches *n + 1* frames (plus `main`), so memory is *O(n)*.

Common mistake

Writing `a[n]` in the combination step goes past the end of the array on the very first call (`n = a.length`) and throws `ArrayIndexOutOfBoundsException`; and since the base case is `n == 0`, `a[0]` would never be added. The last of the first *n* elements is `a[n - 1]`.

The same pattern handles other properties, such as the largest element. Its base case is *n = 1*, where the answer is `a[0]` (an empty part has no largest element); the step compares the largest of the first *n – 1* with the last: `Math.max(largest(a, n - 1), a[n - 1])`. Again *O(n)* time and *O(n)* stack.

4.3.3 Reversing an array

To reverse an array in place, swap the two end elements, then reverse the part between them, which is two elements shorter.

The part is given by its first and last indices: `a[i..j]` is `i` through `j`, inclusive. The first call is `reverse(a, 0, n - 1)`.

- **Base case:** if `i >= j`, 0 or 1 elements remain, and such a part is its own reverse.
- **Reduction and combination:** swap `a[i]` and `a[j]`, then reverse the inner part `a[i+1..j-1]`.

```
void reverse(int[] a, int i, int j) {
  if (i >= j)
    return;                    // base: 0 or 1 elements
  int t = a[i]; a[i] = a[j]; a[j] = t;  // the ends
  reverse(a, i + 1, j - 1);    // the inner part
}
// in main:
int[] a = {2, 7, 4, 9, 1, 6};
reverse(a, 0, a.length - 1);
```

Two things are new. The method returns nothing (`void`); it changes the array itself, and every call sees the same array. And the work comes *before* the call, so nothing is left for the way back.

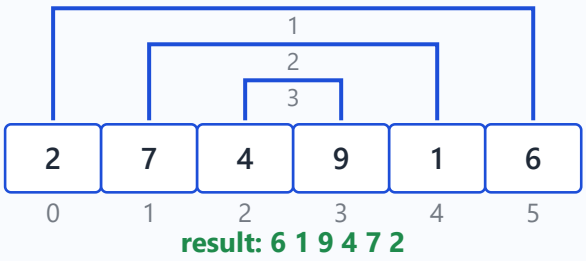


Figure 4.8: Reversing `{2, 7, 4, 9, 1, 6}` : swap elements 0 and 5, then 1 and 4, then 2 and 3. Result: `6 1 9 4 7 2` .

Each call shortens the part by 2, so n elements need $\lfloor n/2 \rfloor$ swaps ($\lfloor\ \rfloor$ rounds down), plus one final call that hits the base case: $\lfloor n/2 \rfloor + 1$ calls. Arrays of 6 and of 7 elements both take 3 swaps and 4 calls (the middle of 7 stays put). Time and stack memory are $O(n)$.

4.3.4 Code trace: reverse(a, 0, 5)

Tracing `reverse(a, 0, 5)` on `a = {2, 7, 4, 9, 1, 6}` , with each call's `i` , `j` , and the array after its swap:

Call	`i` , `j`	Event	Array
—		start	`2 7 4 9 1 6`
1	0, 5	$0 < 5$: `a[0]` ↔ `a[5]` , then `reverse(a, 1, 4)`	`6 7 4 9 1 2`
2	1, 4	$1 < 4$: `a[1]` ↔ `a[4]` , then `reverse(a, 2, 3)`	`6 1 4 9 7 2`
3	2, 3	$2 < 3$: `a[2]` ↔ `a[3]` , then `reverse(a, 3, 2)`	`6 1 9 4 7 2`
4	3, 2	$3 \geq 2$: base case, returns	`6 1 9 4 7 2`

Table 4.4: Tracing `reverse(a, 0, 5)` .

During call 4 the stack holds `main` and 4 `reverse` frames. Then calls 4, 3, 2, 1 return with nothing left to do, since the recursive call is the last line. Result: `6 1 9 4 7 2`, 3 swaps, 4 calls.

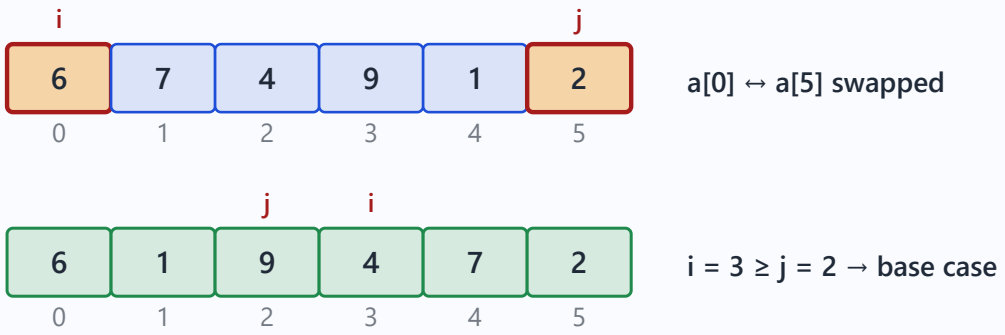


Figure 4.9: Two moments in tracing `reverse(a, 0, 5)`. Top: the first swap gives `6 7 4 9 1 2`. Bottom: the base case, *i = 3, j = 2*; the indices have crossed and every element is in place (green).

Common mistake

The base case `i == j` works only for odd lengths, where the indices meet in the middle. In a 6-element array, after call 3, `i = 3` and `j = 2`; the test fails, the swap repeats, and the array comes apart. The calls go on to `(4, 1)`, `(5, 0)`, `(6, -1)`, and the last accesses `a[6]`: `ArrayIndexOutOfBoundsException`. Use `i >= j`.

4.3.5 Powers, slow and fast

x^n is x multiplied by itself n times, which gives a recursion at once: $x^n = x \cdot x^{n-1}$, $x^0 = 1$.

```
long power(long x, int n) {
  if (n == 0) return 1;
  return x * power(x, n - 1);
}
```

Correct but slow: n drops by only 1 per call, so `power(x, n)` makes $n + 1$ calls, with $O(n)$ time and stack. For $n = 1{,}000{,}000$ the stack overflows.

A faster way uses $x^{10} = (x^5)^2$: compute x^5 once and square it, halving the exponent. An odd exponent sets one x aside: $x^5 = x \cdot (x^2)^2$. In general:

> $x^n = (x^{n/2})^2$ (n even) · $x^n = x \cdot (x^{(n-1)/2})^2$ (n odd)

Java's integer division rounds down, so for odd n, `n / 2` is $(n - 1)/2$, and both cases use `n / 2`. For example:

> $2^{10} = (2^5)^2,\ \ 2^5 = 2 \cdot (2^2)^2,\ \ 2^2 = (2^1)^2,\ \ 2^1 = 2 \cdot (2^0)^2$

```
long fastPower(long x, int n) {
  if (n == 0) return 1;           // base case
  long y = fastPower(x, n / 2);   // x^(n/2)
  if (n % 2 == 0) return y * y;   // n even
  return x * y * y;               // n odd
}
// in main:
System.out.println(fastPower(2, 10));
```

The key is `y` on line 3: $x^{n/2}$ is computed once and squared as `y * y`. Writing `fastPower(x, n / 2) * fastPower(x, n / 2)` would compute it twice and lose the gain (see 4.4.3). Line 4 handles an even exponent; line 5, reached only for an odd one, multiplies by one more x.

Each call halves n, and $\lfloor \log_2 n \rfloor$ halvings bring it to 1; adding the calls for 1 and for the base case 0 gives $\lfloor \log_2 n \rfloor + 2$ calls (for $n \geq 1$). Time and stack are $O(\log n)$.

n	`power` : calls	`fastPower` : calls
10	11	5
100	101	8
1,000	1,001	11
10^6	1,000,001	21

Table 4.5: Number of calls made by `power` and `fastPower`.

Table 4.5 makes the two growth rates concrete. When n grows a thousandfold, `power` makes a thousand times as many calls, `fastPower` only about 10 more. For an exponent of a million, `power` overflows while `fastPower` needs 21 frames. Shrinking by a constant factor, not a constant amount, makes the difference.

Common mistake

Few calls don't prevent overflow of the result. A `long` holds at most about $9.2 \cdot 10^{18}$, so 2^{63} and beyond come out silently wrong. In practice fast exponentiation is usually done modulo some `m`, taking `% m` after each multiplication.

4.3.6 Code trace: fastPower(2, 10)

On the way down only the exponent is halved; every multiplication happens on the way back.

Step	Call	Event	Returned
1	`fastPower(2, 10)`	$n \neq 0$: first `y = fastPower(2, 5)` ($10 / 2 = 5$)	
2	`fastPower(2, 5)`	first `y = fastPower(2, 2)` ($5 / 2 = 2$)	
3	`fastPower(2, 2)`	first `y = fastPower(2, 1)` ($2 / 2 = 1$)	
4	`fastPower(2, 1)`	first `y = fastPower(2, 0)` ($1 / 2 = 0$)	
5	`fastPower(2, 0)`	base case: $2^0 = 1$	1
6	`fastPower(2, 1)`	$y = 1$; $n = 1$ odd: $2 \cdot 1 \cdot 1$	2
7	`fastPower(2, 2)`	$y = 2$; $n = 2$ even: $2 \cdot 2$	4
8	`fastPower(2, 5)`	$y = 4$; $n = 5$ odd: $2 \cdot 4 \cdot 4$	32
9	`fastPower(2, 10)`	$y = 32$; $n = 10$ even: $32 \cdot 32$	1024

Table 4.6: Tracing `fastPower(2, 10)`. Rows 1–4 descend, row 5 is the base case, rows 6–9 return.

The result, 1024, takes 5 calls; `power(2, 10)` would take 11. The returned values 1, 2, 4, 32, 1024 are 2^0, 2^1, 2^2, 2^5, 2^{10}, and the exponents 0, 1, 2, 5, 10 are the prefixes of `1010`, 10 in binary (`1`, `10`, `101`, `1010`); the odd-exponent steps match its 1 digits.

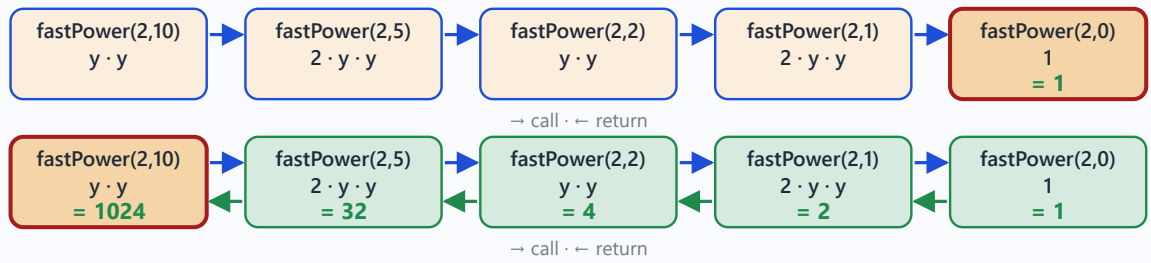


Figure 4.10: The call chain of `fastPower(2, 10)`. Top: the deepest moment (Table 4.6, step 5); each waiting box shows its work on return (`y · y` for an even exponent, `2 · y · y` for odd). Bottom: 1, 2, 4, 32 have come back, and the first call computes $32 \cdot 32 = 1024$.

Example 4.4

How many calls does `fastPower(3, 6)` make, and what does it return?

1. Descent: $n = 6, 3, 1, 0$, so 4 calls, as the formula $\lfloor \log_2 6 \rfloor + 2 = 4$ predicts.
2. `fastPower(3, 0)` returns 1.
3. `fastPower(3, 1)` : n odd, $3 \cdot 1 \cdot 1 = 3$.
4. `fastPower(3, 3)` : n odd, $3 \cdot 3 \cdot 3 = 27$.
5. `fastPower(3, 6)` : n even, $27 \cdot 27 = 729$.

The result is $3^6 = 729$; the slow version would make 7 calls.

4.4 Binary recursion

When a call calls itself twice, two branches grow from each call and the calls form a tree. The best-known example is Fibonacci; this section covers what its tree costs and how to avoid that cost. Binary search, at the end, also has two calls in its code, but its calls form a chain.

4.4.1 Fibonacci numbers

The Fibonacci sequence starts 0, 1, and each later number is the sum of the previous two: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, … It turns up in nature (sunflower seeds, leaf arrangements) and in computer science (the analysis of some data structures).

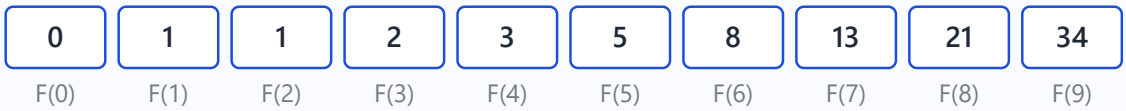


Figure 4.11: The first ten Fibonacci numbers, *F(0)* to *F(9)*. Each is the sum of the two before it, e.g. *F(7) = 8 + 5 = 13*.

The definition is recursive by nature:

> $F(0) = 0,\quad F(1) = 1,\quad F(n) = F(n-1) + F(n-2)\ (n \geq 2)$

Two base cases are needed because the step reaches back two terms; with only *F(0)*, *F(1) = F(0) + F(−1)* would be undefined. In code:

```
int fib(int n) {
  if (n <= 1) return n;    // base case
  return fib(n - 1) + fib(n - 2);
}
```

Line 2 catches both base cases, since $F(n) = n$ for $n \leq 1$. Line 3 makes two recursive calls and adds the results.

Definition 4.4

In **binary recursion** a call makes two recursive calls; with more than two it is **multiple recursion**. The calls then form a **call tree**: each node is a call, and its children are the calls it makes.

The call tree is the basic tool for analyzing binary recursion: its node count is the total number of calls, and its depth the most frames on the stack at once. In the Fibonacci tree the same value is computed many times, and this repetition sets the cost of `fib`.

4.4.2 Code trace: the call tree of fib(4)

```
int fib(int n) {
  if (n <= 1) return n;    // base case
  return fib(n - 1) + fib(n - 2);
}
// in main:
System.out.println(fib(4));
```

The rule to remember: Java evaluates left to right, so in `fib(n - 1) + fib(n - 2)` the call `fib(n - 1)` finishes *completely* before `fib(n - 2)` starts.

Call	Called by	Event	Returned
1: `fib(4)`	`main`	*n > 1*: first `fib(3)`; `fib(2)` waits	
2: `fib(3)`	1, left	first `fib(2)`; `fib(1)` waits	
3: `fib(2)`	2, left	first `fib(1)`; `fib(0)` waits	
4: `fib(1)`	3, left	base case	1
5: `fib(0)`	3, right	base case	0
3: `fib(2)`		*1 + 0*	1
6: `fib(1)`	2, right	base case	1
2: `fib(3)`		*1 + 1*	2
7: `fib(2)`	1, right	computed again: first `fib(1)`	
8: `fib(1)`	7, left	base case	1
9: `fib(0)`	7, right	base case	0
7: `fib(2)`		*1 + 0*	1
1: `fib(4)`		*2 + 1*	3

Table 4.7: Tracing `fib(4)`. Calls are numbered in starting order.

F(4) = 3 takes 9 calls: one each for `fib(4)` and `fib(3)`, two for `fib(2)`, three for `fib(1)`, two for `fib(0)`. Call 7 is pure waste: call 3 already found `fib(2)`, but nothing stored it.

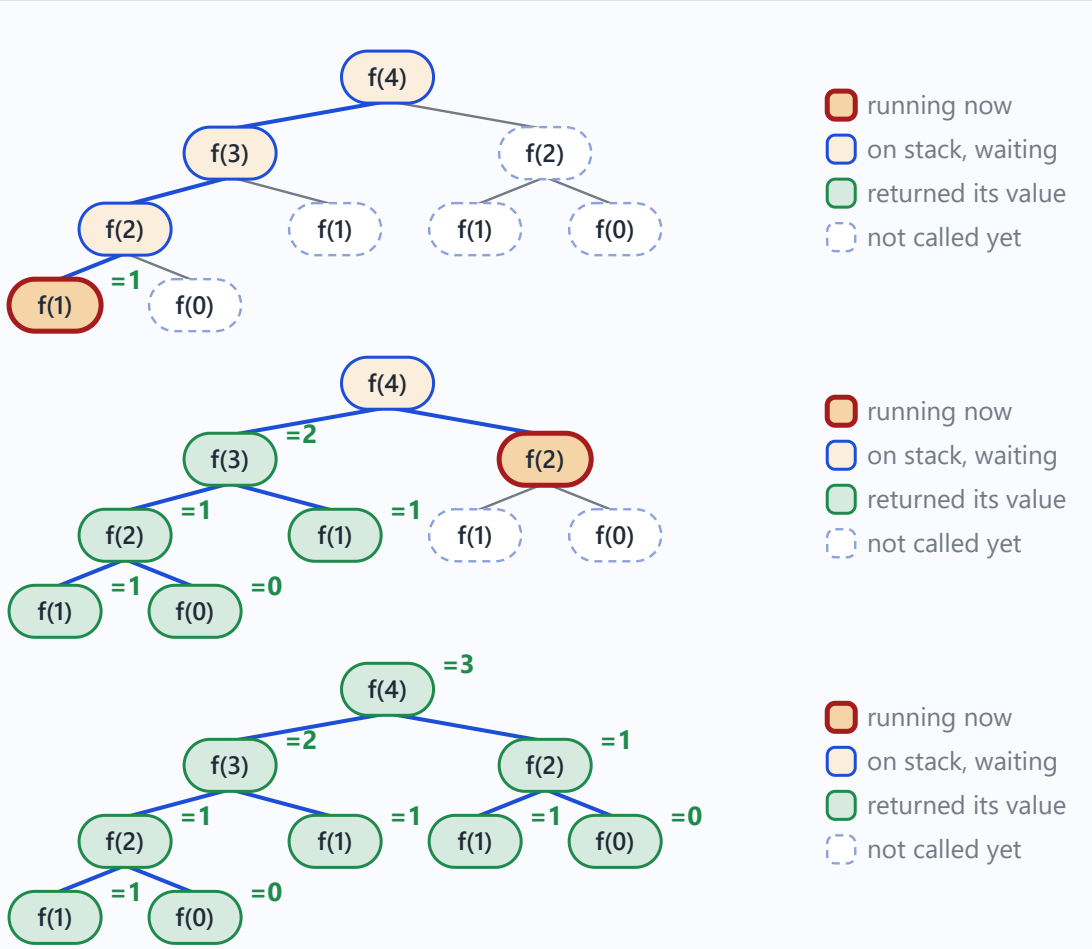


Figure 4.12: Three moments in the call tree of `fib(4)`. Top: the first base case; `fib(1)` runs (red outline), `fib(2)`, `fib(3)`, `fib(4)` wait, and dashed nodes are not yet called. Middle: after `fib(3) = 2`, `fib(4)` computes `fib(2)` a second time. Bottom: all 9 calls have returned (green); `fib(4) = 3`.

The tree also gives the memory cost: the frames on the stack are the path from the root to the running node. The longest path, `fib(4)` → `fib(3)` → `fib(2)` → `fib(1)`, has 4 frames (5 with `main`); despite 9 calls, the stack never holds more.

Common mistake

The two recursive calls do not run at the same time. The calling line can't continue until a call finishes, so the second call starts only after the first call's whole subtree is done; the tree is explored depth-first, left to right.

4.4.3 What recursion costs

A recursive method's time is its total work. If each call does constant work, the total is proportional to the number of calls, so finding the cost means counting calls.

For `factorial` this is easy. Let $T(n)$ be the work of `factorial(n)` and c a call's own constant work; then $T(n) = T(n - 1) + c$. The chain is n calls long, so time and stack are $O(n)$. Such equations are **recurrence relations** (Section 4.6).

`fib` is different: each call makes two, and the same subproblems are solved again and again.

```
int fib(int n) {
  if (n <= 1) return n;
  return fib(n - 1) + fib(n - 2);
}
```

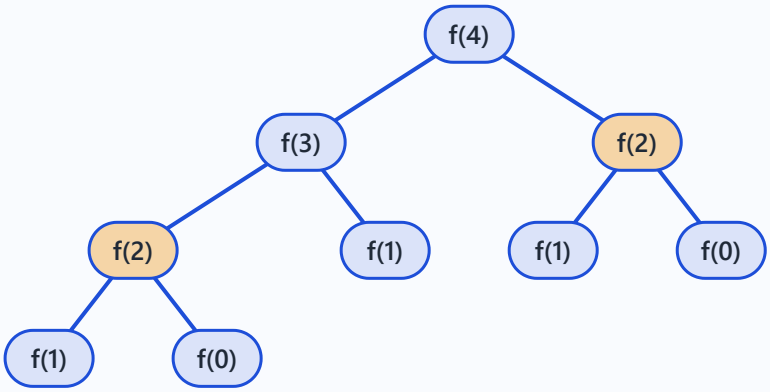


Figure 4.13: The call tree of `fib(4)`, 9 nodes. The orange `f(2)` nodes compute the same subproblem twice, each calling `f(1)` and `f(0)` again.

The call count grows very fast. Let $C(n)$ be the calls made by `fib(n)`. For $n \le 1$, $C(0) = C(1) = 1$; otherwise we count the call itself plus both subtrees: $C(n) = C(n - 1) + C(n - 2) + 1$. This resembles the Fibonacci recurrence, and so does its solution.

Theorem 4.1 (Number of Fibonacci calls)

The recursive `fib(n)` makes $C(n) = 2 \cdot F(n + 1) - 1$ calls in total.

Proof. By induction. For $n = 0$ and $n = 1$, $2 \cdot F(1) - 1 = 2 \cdot F(2) - 1 = 1$, as required. For $n \ge 2$, assuming the formula for $n - 1$ and $n - 2$, $C(n) = (2F(n) - 1) + (2F(n - 1) - 1) + 1 = 2(F(n) + F(n - 1)) - 1 = 2F(n + 1) - 1$. □

For `fib(4)` this gives $2 \cdot F(5) - 1 = 9$, matching the trace. Fibonacci numbers grow by a factor of about 1.6 per step (approaching the golden ratio $\varphi \approx 1.618$), so the calls grow like 1.6^n: **exponential** growth, the work multiplying by about 1.6 each time n grows by 1.

Memory is still $O(n)$: however large the tree, the stack holds one root-to-node path, at most n long. `fib`'s problem is time, not memory.

Common mistake

Writing the same recursive call twice in one expression is costly. With `fastPower(x, n / 2) * fastPower(x, n / 2)` instead of `y * y`, the chain becomes a binary tree of depth $log_2\ n$ with about $2^{log\ n} = n$ nodes, and *O(log n)* degrades to *O(n)*. Compute a value once and store it.

4.4.4 Avoiding repeated work

Numbers make exponential growth tangible. Table 4.8 gives `fib(n)` and its call count for a few *n*.

n	`fib(n)`	calls
5	5	15
10	55	177
20	6,765	21,891
30	832,040	2,692,537
40	102,334,155	331,160,281

Table 4.8: Number of calls made by the recursive `fib`.

`fib(40)` makes over 331 million calls for just 41 distinct subproblems, `fib(0)` to `fib(40)`. Nearly every call re-solves a solved problem. There are two cures.

1. Memoization. Store each value in an array the first time it is computed, and read it from there when it is needed again.

```
long[] memo = new long[91];     // 0: not computed yet
long fibMemo(int n) {
  if (n <= 1) return n;
  if (memo[n] != 0) return memo[n];
  memo[n] = fibMemo(n - 1) + fibMemo(n - 2);
  return memo[n];
}
```

`memo` holds *F(0)* to *F(90)*, all within `long` range (which ends at *F(92)*). Cells start at 0, meaning "not computed"; this is safe since $F(n) \neq 0$ for $n \geq 2$. After the base case, the method returns a stored value if there is one; otherwise it makes the two calls, stores the result in `memo[n]`, and returns it.

`fibMemo(40)` makes 79 calls. The left call, `fibMemo(n - 1)`, computes and stores every value up to $n - 1$, so the right call, `fibMemo(n - 2)`, finds its value ready and returns at once. Each $k \geq 2$ thus has one computing call and one quick call on its right (or a base case like `fibMemo(0)`), plus the bottom `fibMemo(1)`: $2n - 1$ calls, 79 for $n = 40$. Time is *O(n)*; stack depth and the `memo` array are *O(n)* too.

2. A loop. Computing the sequence from the front needs only the last two terms at each step.

```
long fibLoop(int n) {
  long a = 0, b = 1;              // F(0), F(1)
  for (int i = 0; i < n; i++) {
    long t = a + b;  a = b;  b = t;
  }
  return a;
}
```

`a` and `b` hold two consecutive terms, starting at `F(0)` and `F(1)`. Each iteration computes the next term, `t = a + b`, and slides the window forward; after *n* iterations `a = F(n)`.

`i`	`t = a + b`	`a`	`b`
start		0	1
0	1	1	1
1	2	1	2
2	3	2	3
3	5	3	5
4	8	5	8
5	13	8	13

Table 4.9: Tracing `fibLoop(6)`.

After six iterations it returns `a = 8 = F(6)`. The loop takes *O(n)* time and *O(1)* memory, opens no frames, and can't overflow the stack: the best choice for Fibonacci.

Common mistake

Declaring the memo array inside the method defeats it: each call creates its own empty array, nothing is shared, and the time stays exponential. Keep the array where all calls see it, as a field or a parameter.

4.4.5 Recursive binary search

To find *x* in a sorted array, look at the middle element. If it equals *x*, done. If *x* is smaller, it can only be in the left half; if larger, only in the right half. Either way the same search repeats on half the array — a naturally recursive algorithm.

The part searched is `a[lo..hi]`, with two base cases:

- **Empty range** (`lo > hi`): *x* is not in the array; return −1.
- **Middle element is *x***: return the middle index.

```
int binarySearch(int[] a, int x, int lo, int hi) {
  if (lo > hi) return -1;       // base: empty range
  int mid = (lo + hi) / 2;
  if (a[mid] == x) return mid;  // base: found
  if (x < a[mid])
    return binarySearch(a, x, lo, mid - 1); // left half
  return binarySearch(a, x, mid + 1, hi);   // right half
}
System.out.println(binarySearch(a, 23, 0, 9));
```

The code has two recursive calls (lines 6 and 7), but each call makes only one: left if the `if` holds, right otherwise. The calls form a chain, not a tree, so binary search is *linear* recursion; it appears here to contrast the two shapes.

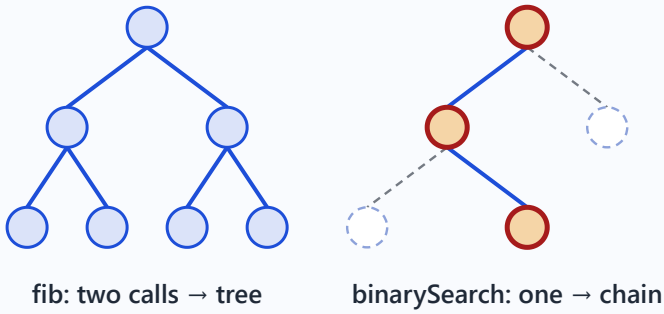


Figure 4.14: Left, `fib`: each call makes two, forming a tree. Right, `binarySearch`: only one half is entered (red path), the other never (dashed), forming a chain.

The range halves each call: $T(n) = T(n/2) + c$, solved by $O(\log n)$ time and stack (4.6.3). Ten elements take at most 5 calls, 10^6 at most 21, counting the final call on an empty range.

Common mistake

Forgetting the `return` before a recursive call loses its result. If line 6 reads `binarySearch(a, x, lo, mid - 1);`, the left half's answer is discarded and execution falls through to the right-half call on line 7. The code still compiles, since the method ends with a `return`, but the answer is wrong. Use or pass on every returned value.

4.4.6 Code trace: binarySearch(a, 23, 0, 9)

Search `a = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91}` for $x = 23$ with `binarySearch(a, 23, 0, 9)`.

Call	`lo, hi`	`mid`	`a[mid]`	Decision
1	0, 9	$(0 + 9)/2 = 4$	16	$23 > 16$: right half, `binarySearch(a, 23, 5, 9)`

Call	`lo, hi`	`mid`	`a[mid]`	Decision
2	5, 9	*(5 + 9)/2 = 7*	56	*23 < 56*: left half, `binarySearch(a, 23, 5, 6)`
3	5, 6	*(5 + 6)/2 = 5*	23	found: returns 5

Table 4.10: Tracing `binarySearch(a, 23, 0, 9)`.

Call 3 returns 5 from the base case; calls 2 and 1 pass it straight on, computing nothing new. The answer takes 3 calls as the range shrinks from 10 to 5 to 2 elements.



Figure 4.15: Two moments in tracing `binarySearch(a, 23, 0, 9)`. Top: call 1, a 10-element range, `mid = 4`, `a[4] = 16`. Bottom: call 3, range `lo = 5`, `hi = 6`, and `a[5] = 23` is found. Elements outside the range are faded.

Example 4.5

What if we search the same array for *x = 50*?

1. `(0, 9)`: `mid = 4`, `a[4] = 16`; *50 > 16*, right half.
2. `(5, 9)`: `mid = 7`, `a[7] = 56`; *50 < 56*, left half.
3. `(5, 6)`: `mid = 5`, `a[5] = 23`; *50 > 23*, right half.
4. `(6, 6)`: `mid = 6`, `a[6] = 38`; *50 > 38*, right half.
5. `(7, 6)`: `lo > hi`, the range is empty: returns −1.

Five calls, the worst case for *n = 10*. The final `lo = 7` is where 50 would be inserted to keep the order, but the method returns only −1.

Common mistake

Bounding the new range by `mid` — `binarySearch(a, x, lo, mid)` or `binarySearch(a, x, mid, hi)` — keeps the already checked `a[mid]` in range. In a two-element range `mid = lo`, the same range is searched forever, and the stack overflows. Use `mid - 1` and `mid + 1`.

4.5 The Towers of Hanoi and backtracking

The Fibonacci tree showed recursion used badly, repeating the same work. Here too the call tree grows exponentially, but every node does *different*, necessary work: in Hanoi each node is a disk move, in backtracking each leaf is a wanted subset or permutation. The output itself is exponential, so the cost is unavoidable, and recursion is the clearest way to produce it.

4.5.1 The Towers of Hanoi

The **Towers of Hanoi** is a classic puzzle. Rod A holds n disks, largest at the bottom; the goal is to move them all to rod C, using rod B as a helper, under two rules:

- Each move takes one disk, the top disk of some rod.
- A disk may be placed only on a larger disk or on an empty rod.

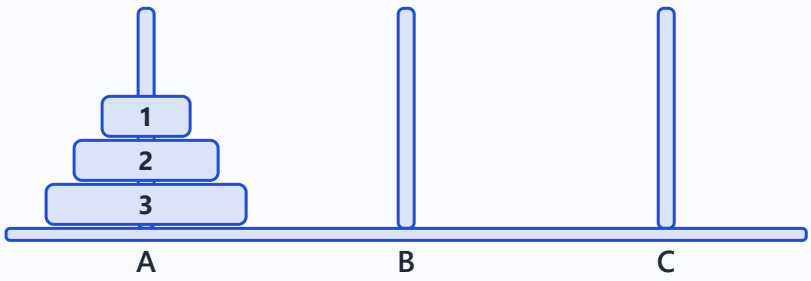


Figure 4.16: The Towers of Hanoi with three disks, all starting on A (3 largest, 1 smallest). The goal is to move them to C.

Move by move, the puzzle is hard; recursively, it nearly solves itself. Consider the largest disk. At some point it moves from A to C, and at that moment A can hold nothing above it and C must be empty, so the other $n - 1$ disks must all be on B. Hence a three-step plan:

1. Move the top $n - 1$ disks from A to B (using C as the helper).
2. Move the largest disk from A to C.
3. Move the $n - 1$ disks from B to C (using A as the helper).

Steps 1 and 3 are the same problem with $n - 1$ disks and the rods in different roles. The largest disk is never disturbed meanwhile, and any disk may sit on it, so for the smaller problem it might as well not exist. The base case, one disk, moves straight from source to target.

4.5.2 Seven moves for three disks

For three disks: 2 disks from A to B (3 moves), disk 3 from A to C, then 2 disks from B to C (3 moves) — 7 moves in all.

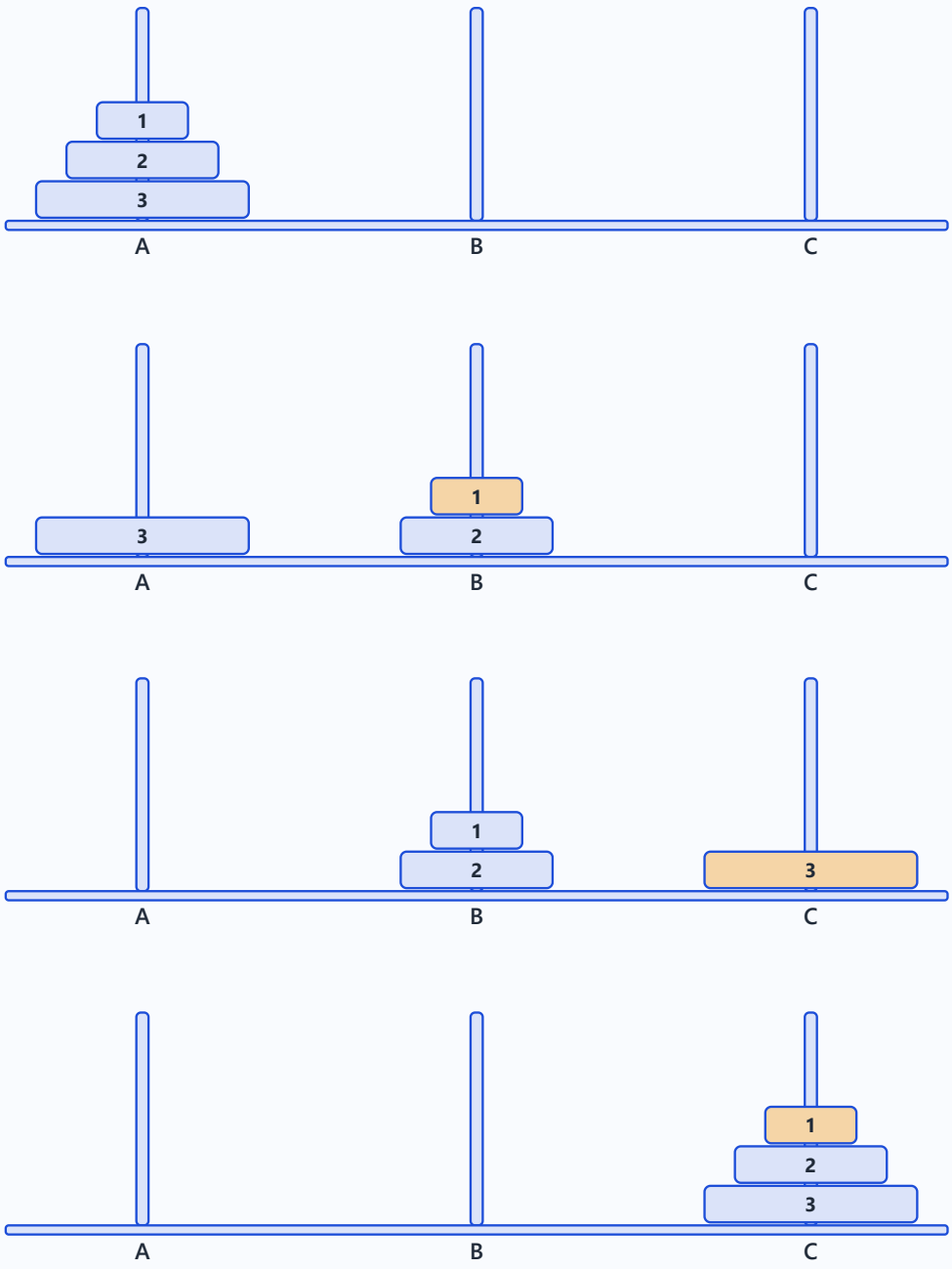


Figure 4.17: Four moments in the seven moves, top to bottom: the start; after move 3 (small disks on B); after move 4 (disk 3 on C); after move 7 (all on C). The moves: disk 1, A → C; disk 2, A → B; disk 1, C → B; disk 3, A → C; disk 1, B → A; disk 2, B → C; disk 1, A → C. The last disk moved is orange.

Figure 4.17 shows the plan's three steps: moves 1–3 gather the small disks on B, move 4 moves the largest disk, and moves 5–7 stack the small disks on it at C. Moves 1–3 and 5–7 are the same job, "move 2 disks," with different rods.

Let $T(n)$ be the moves needed for n disks. Then $T(1) = 1$ and, from the plan, $T(n) = 2T(n - 1) + 1$, giving $T(2) = 3$, $T(3) = 7$, $T(4) = 15$ — each one less than a power of two.

Theorem 4.2 (Number of Hanoi moves)

The Towers of Hanoi with n disks take exactly $2^n - 1$ moves: the recursive plan uses that many, and no solution uses fewer.

Proof. *Sufficiency.* $T(1) = 1 = 2^1 - 1$, and if $T(n - 1) = 2^{n-1} - 1$, then $T(n) = 2(2^{n-1} - 1) + 1 = 2^n - 1$.

Necessity. The largest disk must move at least once. Before its first move, the other $n - 1$ disks must be gathered on one rod: at least $T(n - 1)$ moves. After its last move to C, they must move onto C: at least $T(n - 1)$ more. With at least one move of the largest disk between, the total is at least $2T(n - 1) + 1$. □

$2^n - 1$ grows fast: 20 disks need 1,048,575 moves. Legend has temple priests moving a 64-disk tower, the world ending when they finish. That is about $1.8 \cdot 10^{19}$ moves, some 585 billion years at one move per second.

4.5.3 Code trace: hanoi(3, 'A', 'B', 'C')

The plan goes straight into Java, with parameters `n` (disks), `from` (source), `via` (helper), and `to` (target).

```
void hanoi(int n, char from, char via, char to) {
  if (n == 1) {
    System.out.println("1: " + from + "→" + to);
    return;
  }
  hanoi(n - 1, from, to, via);
  System.out.println(n + ": " + from + "→" + to);
  hanoi(n - 1, via, from, to);
}
// in main:
hanoi(3, 'A', 'B', 'C');
```

Lines 2–5, the base case, move one disk and print, say, "`1: A→C`", the number being the disk. Line 6 is step 1, moving $n - 1$ disks to the helper; note that target and helper swap roles: `hanoi(n - 1, from, to, via)`. Line 7 moves disk `n`. Line 8 is step 3, moving $n - 1$ disks from the helper to the target with the source as helper: `hanoi(n - 1, via, from, to)`.

Step	Running call	Event	Printed
1	`H(3,A,B,C)`	*n* ≠ *1*; line 6: first 2 disks from A to B	
2	`H(2,A,C,B)`	*n* ≠ *1*; line 6: first 1 disk from A to C	
3	`H(1,A,B,C)`	base case	`1: A→C`
4	`H(2,A,C,B)`	line 7: disk 2	`2: A→B`
5	`H(1,C,A,B)`	called from line 8; base case	`1: C→B`
6	`H(3,A,B,C)`	`H(2,A,C,B)` done; line 7: disk 3	`3: A→C`
7	`H(2,B,A,C)`	called from line 8; line 6: first 1 disk from B to A	
8	`H(1,B,C,A)`	base case	`1: B→A`
9	`H(2,B,A,C)`	line 7: disk 2	`2: B→C`
10	`H(1,A,B,C)`	called from line 8; base case	`1: A→C`
11	`H(3,A,B,C)`	subcalls done; returns to `main`	

Table 4.11: Tracing `hanoi(3, 'A', 'B', 'C')`; `H(n,from,via,to)` is a call of `hanoi`.

The seven lines printed are the seven moves of Figure 4.17. The call tree has 7 nodes — one `H(3,…)`, two `H(2,…)`, four `H(1,…)` — each printing one line, so calls equal moves: $2^3 - 1 = 7$.

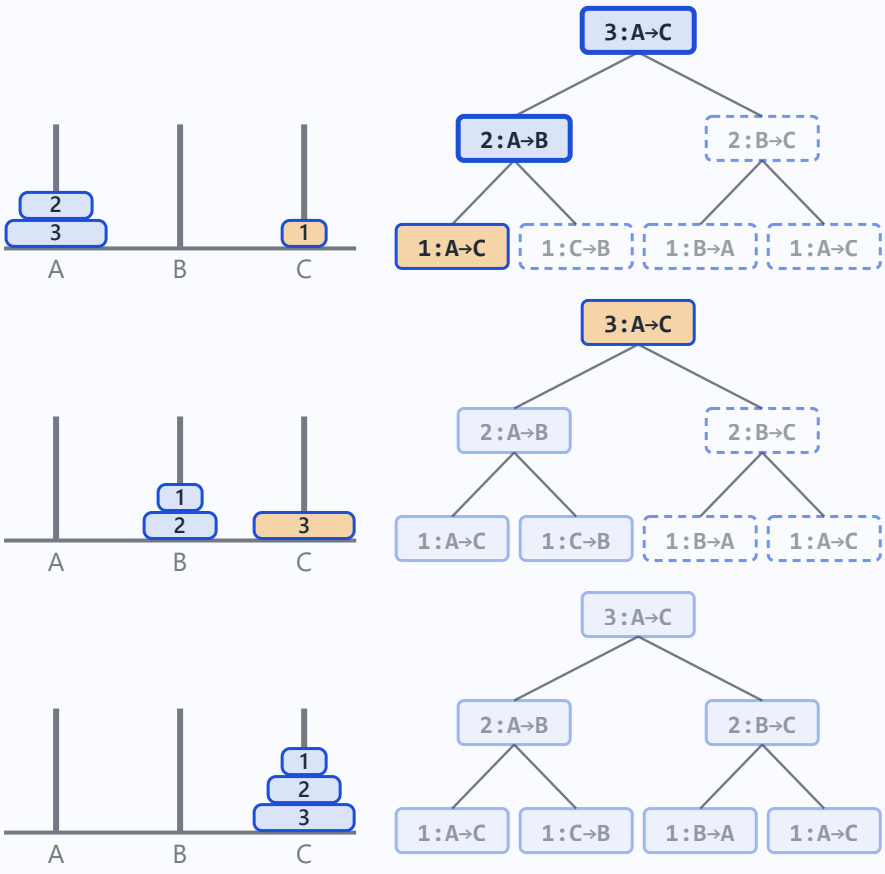


Figure 4.18: The call tree of `hanoi(3, 'A', 'B', 'C')` with the rods. Each box is a call, labeled with the move it prints (`n:source→target`). Top: after the first move (`1:A→C`); thick boxes are on the stack. Middle: the left

subtree is done, the small disks are on B, and disk 3 has moved to C. Bottom: all calls done, all disks on C.

Time is $O(2^n)$, since each of the $2^n - 1$ calls does constant work. Memory is $O(n)$: at most n `hanoi` frames (for n, ..., 1 disks), $n + 1$ with `main`.

Common mistake

Mixing up the rods breaks the solution. With `hanoi(n - 1, from, via, to)` on line 6, the small disks go to the target, and the largest must then land on smaller ones. For each call, ask "from where, to where, and which rod stays free?" In step 1, $n - 1$ disks go from `from` to `via`, and `to` stays free.

There is another pattern: disk 1 moves on every odd-numbered move, 2^{n-1} times in all, and in general disk k moves 2^{n-k} times, the largest just once — the node counts of the tree's levels. In the Towers of Hanoi simulator you can solve the puzzle by hand with 1–6 disks or step through the shortest solution, and compare your move count with the minimum ($2^n - 1$; 63 for 6 disks). Each tower acts as a stack: only the top disk can be taken.

4.5.4 Backtracking

Some answers are the result of a series of choices: which way to turn at each junction of a maze, which digit to put in each empty sudoku square, which elements of a set to pick. The general method for such problems is **backtracking**.

Definition 4.5

Backtracking builds a solution step by step. Each step tries an option and moves on with a recursive call; when the call returns, the choice is **undone** and the next option tried. A branch that clearly cannot lead to a solution is abandoned.

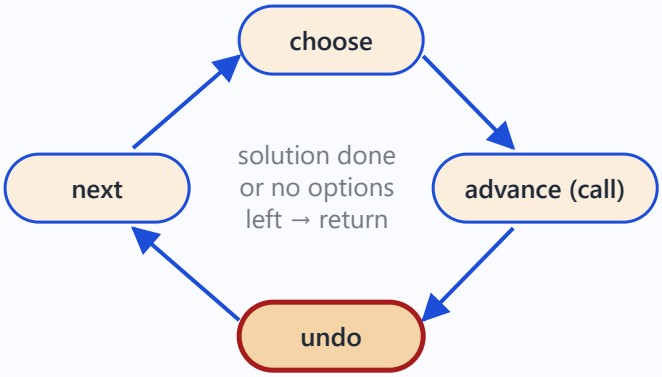


Figure 4.19: The backtracking cycle: choose, advance with a recursive call, undo on return, try the next option. The call returns when the solution is complete or no options remain.

Why undo? The solution under construction (a list, say) is usually one object shared by all calls. A change left in place by one branch would make the next branch start from the wrong state; undoing it lets every branch start from the same state.

As in Hanoi, each call makes several recursive calls. The call tree of all choices tried is a **decision tree**, each root-to-leaf path one sequence of choices. Classic examples: listing subsets or permutations, finding a path through a maze, sudoku, and the n-queens problem (placing n queens on a chessboard so none attacks another).

4.5.5 Listing subsets

To print all subsets of a set — *{1, 2, 3}* has 8: *{1, 2, 3}*, *{1, 2}*, *{1, 3}*, *{1}*, *{2, 3}*, *{2}*, *{3}*, and the empty set — decide for each element in turn whether it is in. Each element has two options, independent of the rest, so n elements give 2^n decision sequences, that is, 2^n subsets.

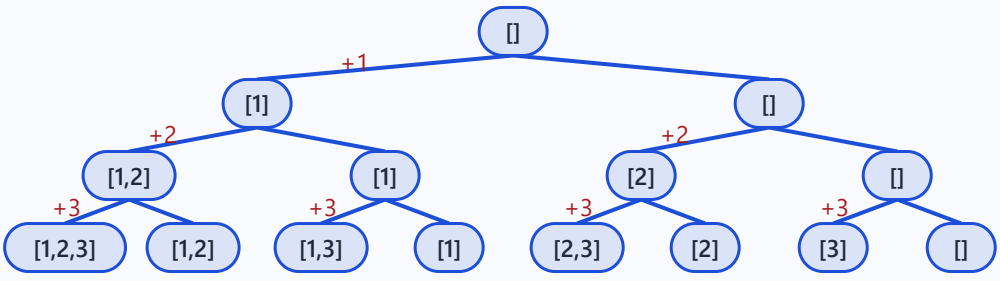


Figure 4.20: The decision tree for *{1, 2, 3}*. Each level decides one element: left takes it (`+1` , `+2` , `+3`), right doesn't. Nodes show the elements chosen so far; the 8 leaves are the 8 subsets.

The tree becomes a method with three parameters: the array `a`, the index `i` to decide, and the list `s` of elements chosen so far.

```
void subsets(int[] a, int i, List<Integer> s) {
  if (i == a.length) {              // base case
    System.out.print(s + " "); return;
  }
  s.add(a[i]);                      // take a[i]
  subsets(a, i + 1, s);
  s.remove(s.size() - 1);           // undo
  subsets(a, i + 1, s);             // try without a[i]
}
// in main:
subsets(new int[]{1, 2, 3}, 0, new ArrayList<>());
```

Each part matches the decision tree:

- **Base case** (lines 2–4): `i == a.length` means every element is decided; we are at a leaf, and `s` is printed.
- **Left branch** (lines 5–6): add `a[i]` and decide the next element.
- **Undo** (line 7): remove `a[i]`, restoring the list to its state at the call's start.
- **Right branch** (line 8): go on without `a[i]`.

`s.remove(s.size() - 1)` removes the *last* element by index, which is always this call's `a[i]`, since subcalls undo their own additions. `s.remove(a[i])` on a `List<Integer>` is a trap: `a[i]` is an `int`, so Java takes it as an index and removes the wrong element.

The decision tree is a complete binary tree with 2^k nodes at level k ($k = 0, \ldots, n$): $2^{n+1} - 1$ calls in all, 15 for $n = 3$. Printing at each leaf costs $O(n)$, so time is $O(n \cdot 2^n)$; the stack depth is $n + 1$, so memory is $O(n)$.

4.5.6 Code trace: subsets of {1, 2, 3}

Table 4.12 lists the 15 calls of `subsets(new int[]{1, 2, 3}, 0, new ArrayList<>())` in starting order, with each call's `i`, its `s` on entry, and its first action.

Call	`i`	`s` on entry	Event
1	0	[]	take 1, go left
2	1	[1]	take 2, go left
3	2	[1, 2]	take 3, go left
4	3	[1, 2, 3]	base case: print **[1, 2, 3]**
5	3	[1, 2]	(call 3 undid 3) base case: **[1, 2]**
6	2	[1]	(call 2 undid 2) take 3
7	3	[1, 3]	base case: **[1, 3]**
8	3	[1]	(3 undone) base case: **[1]**
9	1	[]	(call 1 undid 1) take 2

Call	`i`	`s` on entry	Event
10	2	[2]	take 3
11	3	[2, 3]	base case: **[2, 3]**
12	3	[2]	(3 undone) base case: **[2]**
13	2	[]	(2 undone) take 3
14	3	[3]	base case: **[3]**
15	3	[]	(3 undone) base case: **[]**

Table 4.12: Tracing `subsets` with `a = {1, 2, 3}`.

The output, `[1, 2, 3] [1, 2] [1, 3] [1] [2, 3] [2] [3] []`, lists the leaves left to right, since each call explores "take" fully before "skip." Swapping the branches reverses the order but not the subsets.

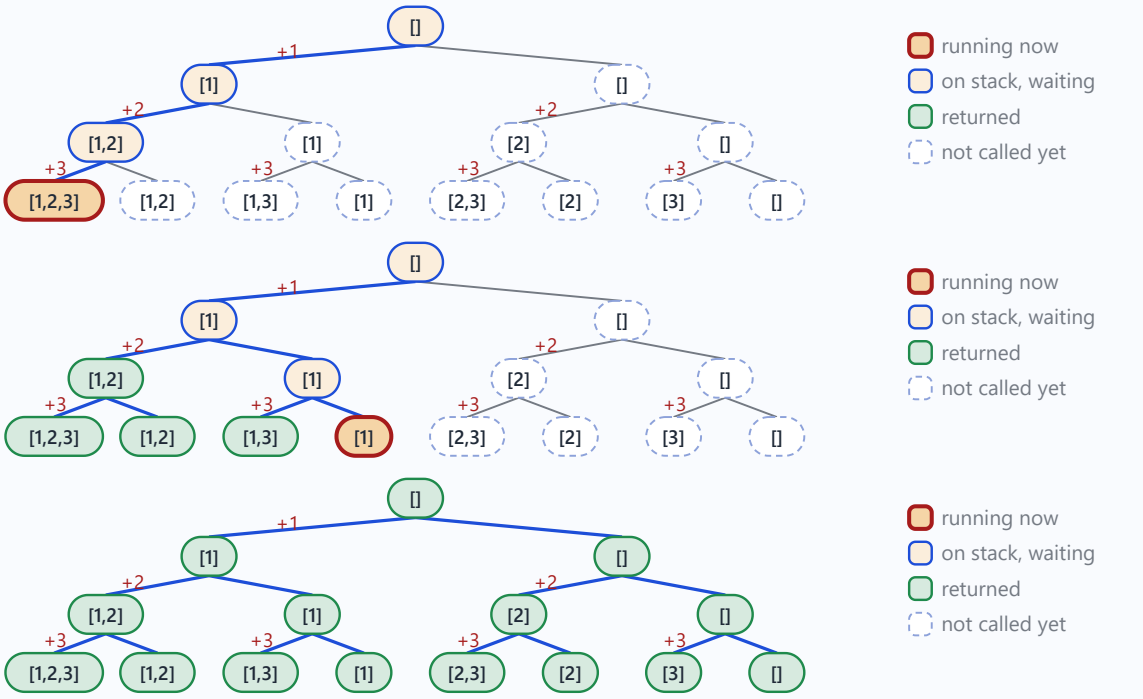


Figure 4.21: Three moments in the decision tree of `subsets`. Top: the first leaf (Table 4.12, call 4) prints `[1, 2, 3]`, its root path on the stack. Middle: call 8 prints `[1]`, finishing the left subtree. Bottom: all 15 calls have returned.

The stack holds at most `main` and 4 `subsets` frames, a root-to-leaf path ($i = 0, 1, 2, 3$).

Common mistake

Without the undo on line 7, an element added in the left branch leaks into the right. For *{1, 2}* the output becomes `[1, 2] [1, 2] [1, 2, 2] [1, 2, 2]`: 2 is added twice, and subsets without 1 never appear. Every change to a shared object must be undone when the call

returns.

4.5.7 Listing permutations

To print every ordering, or **permutation**, of some letters — A, B, C have 6: ABC, ACB, BAC, BCA, CAB, CBA — backtracking picks one of 3 letters first, one of the remaining 2 next, and the last one third: *3 · 2 · 1 = 3! = 6*.

```
void perm(char[] ch, boolean[] used, StringBuilder s) {
  if (s.length() == ch.length) { // base case
    System.out.print(s + " "); return;
  }
  for (int k = 0; k < ch.length; k++) {
    if (used[k]) continue;       // letter already used
    used[k] = true; s.append(ch[k]); // choose
    perm(ch, used, s);
    used[k] = false;             // undo
    s.deleteCharAt(s.length() - 1);
  }
}
```

`ch` holds the letters, `used[k]` records whether `ch[k]` is already in the word, and `s` is the word being built. When the word has every letter (line 2), it is printed. Otherwise each unused letter is tried: mark and append it (line 7), recurse on the rest (line 8), then clear the mark and remove the letter (lines 9–10).

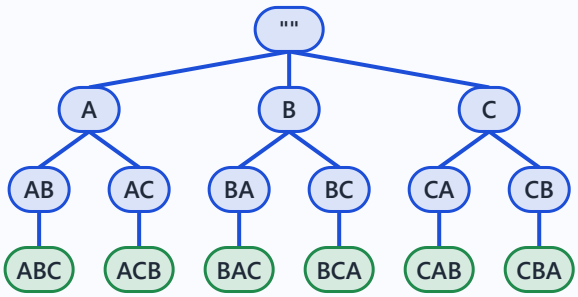


Figure 4.22: The permutation tree for A, B, C. The root word is empty; the levels have 3, 6, and 6 nodes. The leaves (green) are ABC, ACB, BAC, BCA, CAB, CBA: *1 + 3 + 6 + 6 = 16* calls.

A call makes one recursive call per remaining letter — 3 at the root, then 2, then 1 — so the count varies: **multiple recursion**. *n* letters give *n!* leaves and at most about 2.72 · *n!* nodes (16 nodes for *n = 3*). Each node runs an *O(n)* loop and each leaf prints *n* letters, so time is *O(n · n!)*; stack depth is *n + 1*, memory *O(n)*.

Example 4.6

How are the first two permutations of A, B, C produced?

1. Root: `s = ""`. `k = 0`: A is unused; mark it, `s = "A"`, call.
2. `s = "A"`: skip `k = 0` (A is used). `k = 1`: choose B, `s = "AB"`, call.
3. `s = "AB"`: only C is unused; `s = "ABC"`, call.
4. `s = "ABC"`: length 3, print **ABC**, return.
5. In the `s = "AB"` call, C is undone (`s = "AB"`); no letters remain, return.
6. In the `s = "A"` call, B is undone (`s = "A"`). `k = 2`: choose C, `s = "AC"`; then B is appended and **ACB** is printed.

The root then undoes A and tries B first (BAC, BCA), and finally C (CAB, CBA).

Common mistake

Without `used[k] = false;`, a letter used in one branch stays marked in all others. For A, B, C only ABC is printed; later attempts find no unused letters. Since `used` and `s` are shared, both changes must be undone after each attempt.

4.6 Recurrence relations

So far we have counted calls one by one, which works for small inputs but gives no general rule. A better way is to write the cost as an equation and solve it with the call tree. Every method in this chapter fits one of five equation types (Table 4.13).

4.6.1 What is a recurrence relation?

Let $T(n)$ be a method's total work on input size n: its own work plus that of its recursive calls. The latter is the same method's cost on smaller inputs, so $T(n)$ is written in terms of smaller values of T.

Definition 4.6

A **recurrence relation** gives a function's value in terms of its values on smaller inputs. A recursive method's cost $T(n)$ is defined by a base value (such as $T(0) = c$) and a recurrence.

To write one, answer three questions about the method:

1. What does the base case cost? Usually a constant, written c.
2. How many recursive calls are made, and how much smaller are their inputs?

3. How much work does the method itself do outside the calls?

`fastPower`, for example, makes one call on $n / 2$ and otherwise does a comparison and at most two multiplications, so $T(n) = T(n/2) + c$. Table 4.13 lists the chapter's recurrences.

Method	Calls	Recurrence relation	Solution
`factorial`, `sum`, `countdown`	one call, $n - 1$	$T(n) = T(n - 1) + c$	$O(n)$
`reverse`	one call, $n - 2$	$T(n) = T(n - 2) + c$	$O(n)$
`binarySearch`, `fastPower`	one call, $n / 2$	$T(n) = T(n/2) + c$	$O(\log n)$
`hanoi`	two calls, $n - 1$	$T(n) = 2T(n - 1) + c$	$O(2^n)$
`fib`	two calls, $n - 1$ and $n - 2$	$T(n) = T(n - 1) + T(n - 2) + c$	exponential

Table 4.13: Recurrence relations of the methods in this chapter.

The easiest solution method is the call tree. Each node does c work, so $T(n) \approx c \cdot$ (number of calls): solving the recurrence means counting nodes.

Common mistake

Count the calls a call *actually makes*, not those written in the code. `binarySearch` has two calls in its code but makes only one, so its recurrence is $T(n/2) + c$, not $2T(n/2) + c$.

4.6.2 T(n) = T(n − 1) + c

This is linear recursion shrinking the input by 1. We solve it by substituting the recurrence into itself, called **unrolling**:

```
T(n) = T(n - 1) + c
     = T(n - 2) + 2c
     = T(n - 3) + 3c
     …
     = T(0) + n · c
     = c + n · c   →  O(n)
```

Each step adds a c and lowers n by 1, so after k steps $T(n) = T(n - k) + k \cdot c$; at $k = n$, $T(n) = c + n \cdot c$, which is $O(n)$.

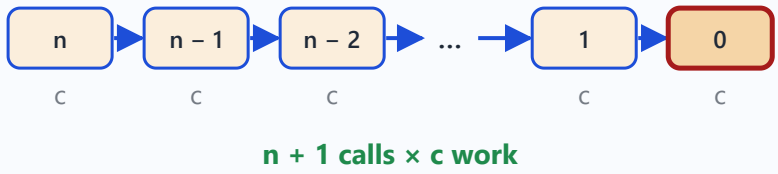


Figure 4.23: The call tree of $T(n) = T(n-1) + c$: a chain of $n + 1$ calls on $n, n-1, \ldots, 0$, each doing c work.

The tree is a chain (Figure 4.23) of $n + 1$ calls, all on the stack at once, since the first can't finish before the last returns; memory is $O(n)$. `factorial`, `sum`, `countdown`, slow powers, and list length fit here.

For `reverse` the input drops by 2: $T(n) = T(n-2) + c$, which unrolls to $T(n-2k) + k \cdot c$ and bottoms out after about $n/2$ steps. Half the calls, but still $O(n)$: shrinking by any constant *amount* gives a linear chain.

4.6.3 T(n) = T(n/2) + c

This is recursion that halves the input. Unrolling:

```
T(n) = T(n/2) + c
     = T(n/4) + 2c
     = T(n/8) + 3c
     …
     = T(n/2^k) + k · c
```

After k steps the input is $n/2^k$, which reaches 1 when $2^k = n$, that is, $k = \log_2 n$. So $T(n) = T(1) + c \cdot \log_2 n$, which is $O(\log n)$.

The **logarithm** here answers one question: how many halvings take n to 1? 16 → 8 → 4 → 2 → 1 takes 4, and $\log_2 16 = 4$. For other n, integer division rounds down, giving $\lfloor \log_2 n \rfloor$ halvings.

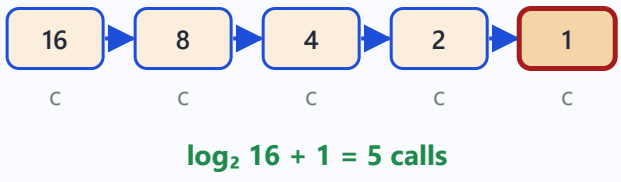


Figure 4.24: The halving chain for $n = 16$: 16, 8, 4, 2, 1, each call doing c work; $\log_2 16 + 1 = 5$ calls.

n	16	1,024	1,000,000
calls	5	11	20

Table 4.14: Length of the halving chain (from n to 1, both ends included).

A thousandfold larger input (1,024 to about a million) adds only about 10 calls: logarithmic growth is that slow. The chain is *log n* deep, so memory is *O(log n)*. Binary search and fast exponentiation fit here. (A base case of 0 or an empty range adds one call, which is why `fastPower` makes $\lfloor \log_2 n \rfloor + 2$.)

Example 4.7

Solve $T(n) = T(n/2) + c$, $T(1) = c$, for $n = 100$ by counting calls, with Java's integer division.

1. The chain is 100, 50, 25, 12, 6, 3, 1 ($25 / 2 = 12$ and $3 / 2 = 1$ round down).
2. 7 calls: $\lfloor \log_2 100 \rfloor = 6$ (as $2^6 = 64 \le 100 < 128 = 2^7$), plus the call for 1.
3. $T(100) = 7c$.

4.6.4 T(n) = 2T(n − 1) + c

Each call makes *two* calls of size $n - 1$, so calls double at each level: 1 at the root, 2 at level 1, 2^k at level k. With base case $n = 1$, levels run from 0 to $n - 1$.

> $1 + 2 + 4 + \ldots + 2^{n-1} = 2^n - 1$ calls → $O(2^n)$

Consecutive powers of two add up to one less than the next power: $1 + 2 + 4 + 8 = 15 = 16 - 1$. In binary it is obvious: `1111` is 15, and adding 1 gives `10000`, 16.

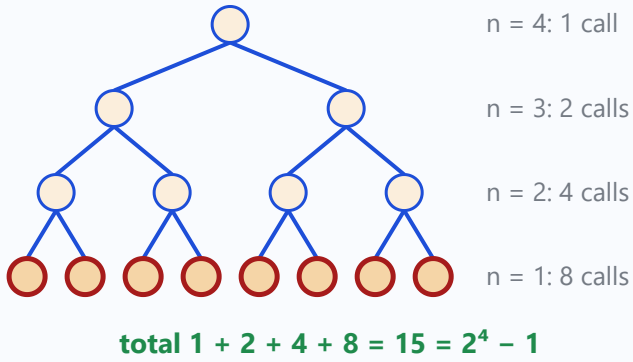


Figure 4.25: The call tree of $T(n) = 2T(n - 1) + c$ for $n = 4$: 1, 2, 4, and 8 calls per level, $15 = 2^4 - 1$ in all. The 8 bottom nodes (red) are base cases.

Hanoi fits exactly: $2^n - 1$ moves, 1,048,575 for $n = 20$, about $1.8 \cdot 10^{19}$ for $n = 64$.

For `fib`, $T(n) = T(n-1) + T(n-2) + c \le 2T(n-1) + c$, since $T(n-2) \le T(n-1)$; so $O(2^n)$ is an upper bound. The true growth, about 1.6^n (4.4.3), is slower but still exponential — hence 331 million calls for $n = 40$.

However large the tree, its depth is only n, so the stack holds at most n frames: exponential time, linear memory.

Common mistake

Don't confuse $2T(n-1)$ with $2T(n/2)$. Both make two calls, but in $T(n) = 2T(n/2) + c$ the tree is $\log_2 n$ deep with $2^{\log n} = n$ leaves, about $2n$ calls or $O(n)$, while $2T(n-1) + c$ is n deep and $O(2^n)$. *How much* the input shrinks matters as much as the number of calls.

4.6.5 Space: the deepest chain of calls

Time comes from *all* the nodes of the call tree; space depends on something else. A call finishes completely before its sibling starts — in `fib(4)`, `fib(2)` waits for the whole subtree of `fib(3)` — so the stack holds only one root-to-node path. Finished calls have no frames left, and unstarted ones none yet.

> Memory = deepest chain of calls × frame size

The depth decides, not the call count: `fib(4)` makes 9 calls but never has more than 4 `fib` frames (plus `main`).

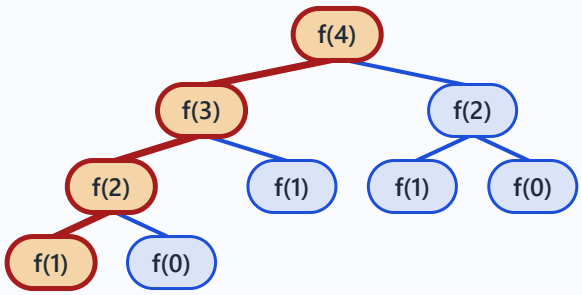


Figure 4.26: The deepest chain (red) in the call tree of `fib(4)`: `fib(4)` → `fib(3)` → `fib(2)` → `fib(1)`. The stack never gets longer.

Call	Total calls	Deepest chain	Time	Space
`factorial(4)`	4	4	$O(n)$	$O(n)$
`fib(40)`	331,160,281	40	exponential	$O(n)$
`hanoi(3, …)`	7	3	$O(2^n)$	$O(n)$

Call	Total calls	Deepest chain	Time	Space
`subsets`, $n = 3$	15	4	$O(n \cdot 2^n)$	$O(n)$

Table 4.15: Total calls versus the deepest chain.

Table 4.15 shows a striking contrast. `fib(40)` makes over 300 million calls, a noticeable wait, yet its stack never exceeds 40 frames. `sumTo(1_000_000)` makes only a million calls, no problem for time, but its million-frame chain overflows the stack (4.2.3). Time and memory problems in recursion are independent; check each separately.

The `subsets` time is not just its $2^{n+1} - 1$ calls: each of the 2^n leaves also prints up to n elements, giving $O(n \cdot 2^n)$.

4.7 Tail recursion

Most recursive methods work on the way back — `factorial` multiplies, `sum` adds, `countUp` prints — so their frames must wait for the subcalls. Methods with nothing to do on the way back are special: they convert easily into loops.

4.7.1 What is tail recursion?

> **Definition 4.7**
>
> A method is **tail recursive** if the recursive call is its last action, with nothing left after the call returns. Such a call is a **tail call**: the method returns its value unchanged.

`factorial` is not tail recursive:

```
int factorial(int n) {
  if (n <= 1) return 1;
  return n * factorial(n - 1);
  // the multiplication happens after the call returns
}
```

The call looks last, but after it returns, its result must still be multiplied by `n`; the multiplication is last.

To make it tail recursive, the multiplication must come *before* the call, by carrying the partial result down in a parameter called an **accumulator**, conventionally `acc`.

```
int factTail(int n, int acc) {
  if (n <= 1) return acc;
  return factTail(n - 1, n * acc);
  // last step: the call (multiplication first)
}
```

`acc` holds the product so far: each call multiplies in its `n` and passes the result on, so at the base case `acc` is the answer. The first call is `factTail(n, 1)`, since nothing has been multiplied yet and 1 is the multiplicative identity.

The chapter's methods split into two groups:

- **Tail recursive:** `countdown` (prints before the call), `reverse` (swaps before the call), `binarySearch` (passes the returned value on unchanged), `factTail`.
- **Not tail recursive:** `factorial` and `sum` (multiply or add after the call), `fib` (adds the results of two calls), `countUp` (prints after the call).

Common mistake

A call on a `return` line isn't necessarily a tail call; what counts is whether work remains after it returns. In `return f(n - 1) + 1;` 1 is added afterward, so this is not tail recursion. `return g(n - 1, acc + n);` is a tail call: the addition happens *before* the call, computing the argument.

4.7.2 Code trace: factTail(4, 1)

```
int factTail(int n, int acc) {
  if (n <= 1) return acc;   // base case
  return factTail(n - 1, n * acc);  // last step
}
// in main:
System.out.println(factTail(4, 1));
```

Step	Call	Event	Returned
1	`factTail(4, 1)`	$n = 4 > 1$: multiply first, $4 \cdot 1 = 4$, then `factTail(3, 4)`	
2	`factTail(3, 4)`	$3 \cdot 4 = 12$, then `factTail(2, 12)`	
3	`factTail(2, 12)`	$2 \cdot 12 = 24$, then `factTail(1, 24)`	
4	`factTail(1, 24)`	base case: returns `acc`; the result is ready here	24
5	`factTail(2, 12)`	passes 24 on unchanged	24
6	`factTail(3, 4)`	passes 24 on unchanged	24

Step	Call	Event	Returned
7	`factTail(4, 1)`	passes 24 on unchanged	24

Table 4.16: Tracing `factTail(4, 1)`.

Compare Table 4.2. There, `factorial(4)` multiplied on the way back ($2 \cdot 1$, $3 \cdot 2$, $4 \cdot 6$); here the multiplications happen on the way down ($4 \cdot 1$, $3 \cdot 4$, $2 \cdot 12$), with the same result since multiplication is commutative. The real difference is the way back: in steps 5–7 every frame just passes 24 on.

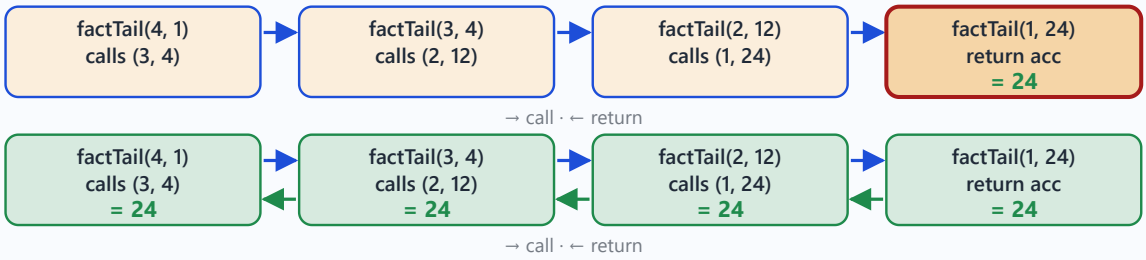


Figure 4.27: The call chain of `factTail(4, 1)`. Top: the base case (Table 4.16, step 4); each box shows the arguments of the call it makes, and `factTail(1, 24)` returns the finished 24. Bottom: every frame has passed 24 on.

With nothing to do on the way back, the waiting frames are useless, and a compiler could reuse the current frame for a tail call: **tail call optimization**. Java doesn't, so `factTail` still opens 4 frames here, needed for nothing yet occupying stack space.

4.7.3 Turning it into a loop

A tail-recursive method always converts mechanically into a loop: a tail call just gives the parameters new values and goes back to the top, which is what a loop does.

```
int factLoop(int n) {
  int acc = 1;
  while (n > 1) {   // while not the base case
    acc = n * acc;  // the call's new
    n = n - 1;      //   arguments
  }
  return acc;       // base: return acc
}
```

The conversion rules are:

- **Parameters become variables.** `acc` becomes a local initialized to its first-call value, 1.
- **The negated base case is the loop condition.** Recursion stops when `n <= 1`; the loop runs while `n > 1`.

- **The tail call becomes updates.** `factTail(n - 1, n * acc)` becomes `acc = n * acc;` and `n = n - 1;`, in that order, since `acc` needs the old `n`.
- **The base case's return follows the loop:** `return acc;`.

Iteration	`n`	`acc`	Corresponds to
start	4	1	`factTail(4, 1)`
1	3	4	`factTail(3, 4)`
2	2	12	`factTail(2, 12)`
3	1	24	`factTail(1, 24)`: condition false, returns 24

Table 4.17: Tracing `factLoop(4)`; each row matches one call of the recursive version.

The loop uses two variables and no frames: *O(1)* memory, *O(n)* time. `binarySearch` converts the same way — `lo` and `hi` become variables, the loop runs while `lo <= hi`, each iteration updates one — which yields the familiar iterative binary search.

Neither the Java compiler nor the JVM optimizes tail calls, so deep tail recursion still throws `StackOverflowError`. Scheme guarantees the optimization, and Scala and Kotlin compilers can turn self-recursive tail calls into loops; in Java, potentially deep tail recursion is converted by hand.

Example 4.8

This tail-recursive method counts digits: `int digits(int n, int acc) { if (n < 10) return acc + 1; return digits(n / 10, acc + 1); }`, first called as `digits(n, 0)`. Convert it to a loop and trace `digits(4096, 0)`.

1. Parameters become variables; start with `acc = 0`.
2. The base case is `n < 10`, so the loop runs while `n >= 10`.
3. The tail call becomes `acc = acc + 1; n = n / 10;` (order is irrelevant here, as neither uses the other's old value).
4. After the loop, `return acc + 1;`.
5. Trace: *(n, acc)* = (4096, 0) → (409, 1) → (40, 2) → (4, 3); now *4 < 10*, so it returns *3 + 1 = 4*.

4.7.4 Recursion or a loop?

Any loop can be written recursively and vice versa; the choice depends on the problem's structure and the input size. Table 4.18 compares the two.

Criterion	Recursion	Loop
Readability	close to the definition, short	state variables managed by hand
Memory	*O*(depth) call stack	usually *O(1)*

Criterion	Recursion	Loop
Speed	every call has overhead	usually faster
Depth limit	risk of `StackOverflowError`	none

Table 4.18: Recursion compared with loops.

Recursion is natural for nested data (trees, folders, expressions), backtracking (subsets, permutations), problems halved at each step, and recursively defined problems like Hanoi. In all of these the depth is small: about *log n*, or *n* itself is small.

A loop is better for simple counting and summing, for linear recursion of depth *n* when *n* may be large (a long list, a big number), and for binary recursion with repeated subproblems (`fib`).

More generally, any recursive method, tail recursive or not, becomes a loop if an explicit stack does the call stack's job: pending work (unfinished multiplications, unsolved subproblems) is pushed, and the loop runs until the stack is empty. Hanoi, for instance, can be solved with such a "task stack." The conversion is longer than for tail recursion but removes the overflow risk, since an explicit stack grows in the program's general memory.

> **Common mistake**
>
> Recursion isn't inherently slow. `fib` is slow because it re-solves subproblems; `binarySearch` and `fastPower` , of depth *O(log n)*, are fast and easy on the stack. Large depth and repetition are the trouble, not recursion.

4.8 Recursion on linked lists

Recursion suits data structures as well as numbers and arrays. Seen recursively, the singly linked list, the simplest linked structure, has very short operations, and the same view pays off for trees.

4.8.1 A list is a recursive structure

A **linked list** stores elements in **nodes**, each holding a value and a reference to the next node; the last node's reference is `null` . The reference to the first node is usually `head` , which is `null` for an empty list.

Recursively, a list is either empty (`null`) or a node followed by *a shorter list*. `head → 7 → 3 → 9 → null` is a node holding 7 plus the list `3 → 9 → null` , which is a node holding 3 plus `9 → null` , and so on down to the empty list.



Figure 4.28: The list `head → 7 → 3 → 9 → null`. Everything from the first node's `next` on (`d.next`) is itself a list: "the rest."

So list operations are naturally recursive, always with one pattern: process the node and pass the rest (`d.next`) to the same method, with the empty list, `d == null`, as base case. Length is the simplest example:

```
class Node { int data; Node next; }

int length(Node d) {
  if (d == null) return 0;        // empty list
  return 1 + length(d.next);      // 1 + the rest
}
```

An empty list has length 0; otherwise the length is 1 plus the rest's. Base case `d == null`, reduction to `d.next`, combination by adding 1. Printing front to back follows the same pattern:

```
void print(Node d) {
  if (d == null) return;
  System.out.print(d.data + " ");
  print(d.next);
}
```

Common mistake

The base case `d.next == null` handles a one-node list but not the empty list: `length(null)` accesses `d.next` and throws `NullPointerException`. The base case is the structure's smallest form, the empty list.

4.8.2 Code trace: length(head)

```
int length(Node d) {
  if (d == null) return 0;        // empty list
  return 1 + length(d.next);      // 1 + the rest
}
// in main (head → 7 → 3 → 9 → null):
System.out.println(length(head));
```

Step	`d`	Event	Returned
1	node 7	$d \neq$ `null` : `1 + length(d.next)` ; the rest of the list starts at 3	
2	node 3	$d \neq$ `null` : the rest starts at 9	
3	node 9	$d \neq$ `null` : `d.next` is `null` ; call on the empty list	
4	`null`	base case: empty list	0
5	node 9	*1 + 0*	1
6	node 3	*1 + 1*	2
7	node 7	*1 + 2*	3

Table 4.19: Tracing `length(head)` on the list `7 → 3 → 9 → null` .

The result is 3, after 4 calls (3 nodes and the final `null`). A list of n nodes takes $n + 1$ calls and as many frames: $O(n)$ time and stack.

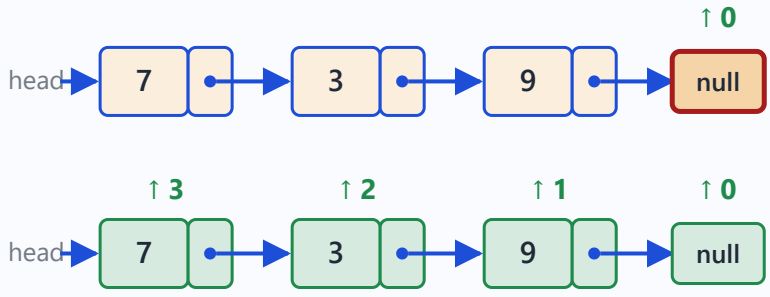


Figure 4.29: Two moments in tracing `length(head)` . Top: the base case (Table 4.19, step 4), `d` is `null` and returns 0. Bottom: above each node, the length of the list starting there (3, 2, 1, 0).

For lists this memory cost is serious: on a million-node list `length` tries to open a million frames, which won't fit (4.2.3). A loop needs constant memory: `int s = 0; for (Node d = head; d != null; d = d.next) s++;` .

4.8.3 Printing a list backward

A singly linked list has no references back, so printing it backward is awkward with a loop: you can reach the last node but can't step back. Recursion solves this by itself.

```
void printReverse(Node d) {
  if (d == null) return;
  printReverse(d.next);   // the rest first
  System.out.print(d.data + " ");
}
```

This differs from `print` (4.8.1) only in whether the print precedes or follows the call — just like `countdown2` and `countUp` (4.1.5). `print(head)` prints `7 3 9`, in visiting order; `printReverse(head)` prints `9 3 7`.



Figure 4.30: The list `head → 7 → 3 → 9 → null`. `print` prints forward (7 3 9); `printReverse` prints backward (9 3 7) as the stack unwinds.

Example 4.9

Tracing `printReverse(head)`:

1. `printReverse(7)`: $d \neq$ `null`; call `printReverse(3)` first; 7 waits.
2. `printReverse(3)`: first `printReverse(9)`; 3 waits.
3. `printReverse(9)`: first `printReverse(null)`; 9 waits.
4. `printReverse(null)`: base case, returns at once.
5. The stack unwinds: `printReverse(9)` prints `9`, then `printReverse(3)` prints `3`, and finally `printReverse(7)` prints `7`.

Output: `9 3 7`, printed back to front as the stack unwinds.

The list has no backward links; the call stack provides the way back, each frame holding its node until its turn, at the cost of $O(n)$ stack. For very long lists, a loop with an explicit stack is safer: push each node while walking the list, then pop and print them in reverse.

Common mistake

Reversing a list just to print it backward changes the structure and breaks any other code using it. `printReverse` only reads the list. To traverse in reverse without changing the structure, use recursion or an explicit stack.

4.9 Common mistakes

Most recursion mistakes fall into a few types. Those noted throughout the chapter are collected here as a checklist for reviewing a recursive method.

Mistake	What happens	The fix
Base case missing or unreachable	An input of 0 or a negative input skips the base case; the calls never end, `StackOverflowError`.	A test that catches every input, such as `n <= 1` instead of `n == 1`.
Problem doesn't shrink	`f(n)` calls `f(n)` or `f(n + 1)`; the calls never approach the base case.	Every call must move the input toward the base case.
Losing the return value	Without the `return` before the left-half call in `binarySearch`, the result is discarded and execution falls through to the right half; the compiler doesn't warn.	Use or return the recursive call's result.
Forgetting to undo	In backtracking, one branch's choices leak into later branches; the subsets or permutations come out wrong.	Undo every change to a shared object when the call returns.
Repeated computation	In binary recursion like `fib`, the same subproblem is solved an exponential number of times.	Memoization or a loop: *O(n)*.
Deep linear recursion	`length` overflows the stack on a list of a million nodes; Java doesn't optimize tail calls.	A loop.

Table 4.20: Common mistakes in recursive code.

All of these come back to two questions. The first is correctness: *does the base case catch every input, and does every step approach it?* The first two mistakes answer “no.” The third and fourth break the induction assumption: the smaller answer is lost, or the smaller problem starts from the wrong state. The second question is cost: *how many nodes does the call tree have, and how deep is it?* The last two mistakes ignore it; exponentially many nodes make time the problem, great depth makes memory the problem.

4.10 Summary

- **Recursion** is a method calling itself on a smaller input. Every recursive method has a **base case**, solved directly, and a **recursive step** that approaches it. Thinking recursively means trusting the smaller problem's answer and designing only the last step — the logic of induction.

- Work **before** the recursive call happens as the calls go down; work **after** it happens as they return, in reverse order (`countdown2` → `3 2 1` , `countUp` → `1 2 3`).
- Each call pushes a **frame** (parameters, locals, return address) onto the **call stack**, in LIFO order; only the top frame runs. The stack is limited, and very deep or infinite recursion throws `StackOverflowError` , so every input, including 0 and negatives, must reach a base case in finitely many steps.
- In **linear recursion** each call makes at most one call, forming a chain: `factorial` , `sum` , `reverse` , `length` . Halving the input gives a chain of length *O(log n)*: `fastPower` makes $\lfloor \log_2 n \rfloor + 2$ calls, and `binarySearch` is *O(log n)*.
- In **binary recursion** each call makes two, forming a **call tree**. `fib(n)` makes $2F(n + 1) - 1$ calls, growing like 1.6^n, because subproblems repeat. Memoization or a loop gives *O(n)*, the loop with *O(1)* memory.
- The **Towers of Hanoi** take three steps: $n - 1$ disks to the helper, the largest to the target, $n - 1$ disks to the target; the $2^n - 1$ moves are the minimum.
- **Backtracking** builds a solution step by step: choose, recurse, and **undo** the choice on return. `subsets` produces 2^n subsets in $2^{n+1} - 1$ calls; `perm` produces *n!* permutations.
- A **recurrence relation** expresses the cost through the cost on smaller inputs and is solved with the call tree: $T(n - 1) + c \to O(n)$, $T(n/2) + c \to O(\log n)$, $2T(n - 1) + c \to O(2^n)$. Time comes from the number of calls, space from the **deepest chain of calls**.
- In **tail recursion** the recursive call is the last action, and an **accumulator** carries the partial result. Such a method converts mechanically into a loop: parameters become variables, the negated base case the loop condition. Java doesn't optimize tail calls, so `factTail` still uses *O(n)* stack.
- A **linked list** is recursive: empty, or a node followed by a shorter list, with base case `d == null` . `printReverse` uses the call stack to print a singly linked list backward, at the cost of *O(n)* stack.

Example	Kind of recursion	Time	Space (stack)
`factorial` , `sum` , `reverse`	linear	*O(n)*	*O(n)*
`fastPower` , `binarySearch`	linear, halving	*O(log n)*	*O(log n)*
`fib`	binary	exponential ($\approx 1.6^n$)	*O(n)*
`hanoi`	binary	$O(2^n)$	*O(n)*
`subsets`	backtracking	$O(n \cdot 2^n)$	*O(n)*
`perm`	multiple, backtracking	$O(n \cdot n!)$	*O(n)*
`factTail`	tail recursion	*O(n)*	*O(n)* (in Java)
`length` , `printReverse`	linear (linked list)	*O(n)*	*O(n)*

Table 4.21: Summary of the recursive methods in this chapter.

In one sentence: recursion is a base case plus a recursive step that approaches it; its time comes from the number of calls, its space from the deepest chain of calls.

4.11 Key terms

Each term links to the course glossary, which gives its Turkish equivalent and its use in other chapters.

recursion · base case · recursive step · stack frame · call stack · LIFO (last-in, first-out) · linear recursion · binary recursion · multiple recursion · binary search · Towers of Hanoi · backtracking · recurrence relation · tail recursion · accumulator · linked list · node · time complexity · space complexity

4.12 Test Yourself

1. Given `void printDigits(int n) { if (n == 0) return; printDigits(n / 10); System.out.print(n % 10 + " "); }`, what does `printDigits(472)` print? How many calls are made, and how many frames (counting `main`) are on the stack at most?

   Answer

   The calls are on 472, 47, 4, 0: 4 calls. `printDigits(0)` is the base case and prints nothing. The print follows the call, so printing happens on the way back: 4, then 7, then 2. The output is `4 7 2`, the digits left to right. At the deepest point there are 4 `printDigits` frames plus `main`: 5.

2. How many calls does `fastPower(3, 9)` make? What does each call return?

   Answer

   The values of n are 9, 4, 2, 1, 0: 5 calls ($\lfloor \log_2 9 \rfloor + 2 = 3 + 2$). On the way back: `fastPower(3, 0) = 1`; $n = 1$ odd, $3 \cdot 1 \cdot 1 = 3$; $n = 2$ even, $3 \cdot 3 = 9$; $n = 4$ even, $9 \cdot 9 = 81$; $n = 9$ odd, $3 \cdot 81 \cdot 81 = 19{,}683$. The result is $3^9 = 19{,}683$.

3. How many calls does the recursive `fib(7)` make? What is the largest number of `fib` frames on the stack at once?

   Answer

   Since $F(8) = 21$, there are $2 \cdot 21 - 1 = 41$ calls, and `fib(7) = 13`. The deepest chain is `fib(7)` → `fib(6)` → … → `fib(1)`: 7 frames (8 with `main`). Time is exponential, memory $O(n)$.

4. For `a = {1, 4, 7, 10, 13, 16, 19}`, how many calls do `binarySearch(a, 19, 0, 6)` and `binarySearch(a, 5, 0, 6)` each make, and what do they return?

   Answer

   For 19: `(0, 6)` `mid = 3`, *19 > 10*; `(4, 6)` `mid = 5`, *19 > 16*; `(6, 6)` `mid = 6`, found. 3 calls, returns 6. For 5: `(0, 6)` *5 < 10*; `(0, 2)` `mid = 1`, *5 > 4*; `(2, 2)` `mid = 2`, *5 < 7*; `(2, 1)` empty range. 4 calls, returns −1.

5. How many moves does the Towers of Hanoi take with 5 disks? How many times does disk 2 move?

   Answer

   $2^5 - 1 = 31$ moves. Disk k moves 2^{n-k} times, so disk 2 moves $2^3 = 8$ times. Check: disks 1 through 5 move 16, 8, 4, 2, and 1 times, which add up to 31.

6. What does `subsets(new int[]{5, 6}, 0, new ArrayList<>())` print, and how many calls does it make?

   Answer

   The output is `[5, 6] [5] [6] []`. The decision tree has 1 + 2 + 4 = 7 nodes: $2^{2+1} - 1 = 7$ calls. The stack holds at most 3 `subsets` frames.

7. A method does constant work and stops when `n <= 0`; otherwise it does constant work and calls itself once on $n - 3$. What is its recurrence relation? How many calls are made for $n = 12$?

   Answer

   $T(n) = T(n - 3) + c$. The chain is 12, 9, 6, 3, 0: 5 calls. In general there are about $n/3 + 1$ calls, which is $O(n)$, since the input shrinks by a constant amount at each step.

8. Is `int pow2(int n, int acc) { if (n == 0) return acc; return pow2(n - 1, acc * 2); }` tail recursive? Through which chain of calls does `pow2(5, 1)` run, and what does it return? What is the loop equivalent?

   Answer

   Yes: the multiplication happens while computing the argument, before the call, and the returned value is passed on unchanged. The chain is *(5, 1) → (4, 2) → (3, 4) → (2, 8) → (1, 16) → (0, 32)*: 6 calls, returning 32 (2^5). As a loop: `int acc = 1; while (n != 0) { acc = acc * 2; n = n - 1; } return acc;`. In Java the recursion uses $O(n)$ memory, the loop $O(1)$.

9. Consider `int count(Node d, int x) { if (d == null) return 0; return (d.data == x ? 1 : 0) + count(d.next, x); }`. For the list `2 → 7 → 2 → 2 → null`, what does `count(head, 2)` return, and how many calls does it make? Is the method tail recursive?

   Answer

   It returns 3 (three nodes hold 2), after 5 calls: four nodes and `null`. It is not tail recursive, since 0 or 1 is added after the call returns; a version with an accumulator (`count(d.next, x, acc + …)`) would be.

10. Given `int t(int n) { if (n <= 1) return 1; return t(n - 1) + t(n - 1) + t(n - 1); }`, what does `t(3)` return, and how many calls does it make? How can the method be made linear?

    Answer

    The tree's levels have 1, 3, and 9 nodes: 13 calls. Each of the 9 leaves returns 1, so `t(3) = 9 = 3²`. In general $t(n) = 3^{n-1}$, with $(3^n - 1)/2$ calls: $O(3^n)$. The three calls compute the same value; `return 3 * t(n - 1);` leaves one call and costs $O(n)$.

4.13 Further reading

- M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, *Data Structures and Algorithms in Java* (6th ed., 2014), Chapter 5: presents recursion by dividing it into linear, binary, and multiple recursion, with Java examples close to those in these notes (binary search, reversing an array, computing powers).
- R. Sedgewick and K. Wayne, *Computer Science: An Interdisciplinary Approach* (2016), Section 2.3: the Towers of Hanoi, repeated computation in Fibonacci, and memoization, through short programs and experiments.
- E. Roberts, *Thinking Recursively with Java* (2006): a whole book on recursive thinking, explaining the connection with induction and backtracking examples for the beginning reader.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms* (4th ed., 2022), Chapter 4: solves recurrences more formally, with the recursion-tree and substitution methods.

4.14 References

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022.
2. M. T. Goodrich, R. Tamassia, M. H. Goldwasser, *Data Structures and Algorithms in Java*, 6th ed., Wiley, 2014.
3. E. Roberts, *Thinking Recursively with Java*, Wiley, 2006.
4. R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach*, Addison-Wesley, 2016.

PART II

Linear Structures

Chapters 5–8

PART II · CHAPTER 5

Linked Lists

An array keeps its elements side by side in memory, which makes indexing fast but forces elements to shift whenever something is inserted at the front or in the middle. A linked list stores them differently: each element sits in its own small box, a node, which knows where the next node is. The links, not memory positions, define the order. We start with nodes, then cover traversal, search, insertion, and deletion in a singly linked list, with code and step-by-step traces, followed by classic problems: reversal, the middle element, copying, removing duplicates, merging sorted lists, and sorting. Doubly linked lists, sentinel nodes, and circular lists come next, and the last section weighs arrays against linked lists and collects the costs.

Code for this chapter: `github.com/sercankulcu/data-structures-java/Bolum05/src`

LEARNING OUTCOMES

Node structure
Explain nodes, links, and references; draw the linked structure that results from a sequence of assignments.

Basic operations
Implement insertion, deletion, search, and traversal in singly, doubly, and circularly linked lists, setting links in the right order.

Kinds of lists
Compare singly, doubly, and circularly linked lists; explain what sentinel nodes and a tail pointer make easier.

Cost analysis
State and justify the time and space cost of linked list operations, and compare it with the cost of the same operation on an array.

5.1 Fundamentals

A program rarely knows in advance how many elements it will hold — how many customers will join a queue, how many lines a user will type, how many objects a game will put on screen. Arrays cope poorly: their capacity is fixed, a full array must be copied into a bigger one, and every insertion at the front or in the middle shifts everything behind it. Linked lists remove both problems, at a price spelled out at the end of this section.

5.1.1 Linked lists and their kinds

Think of a treasure hunt. The first clue says where the second is hidden, the second says where the third is, and so on. The slips are scattered around the house — in a drawer, inside a book, behind a door — yet the order never gets mixed up, because each one points to the next. To play, you only need to know where the first slip is.

A **linked list** works the same way. Its elements are not in adjacent cells; each is stored in a separate object, a **node**, along with the location of the next node. The slip is the node; the address on it is the link.

> **Definition 5.1**
>
> A **linked list** is a linear data structure that stores its elements in nodes connected by references. The order of the elements is set by the links, not by where the nodes sit in memory. The first node is the **head** of the list, the last node its **tail**.

There are three basic kinds, shown side by side in Figure 5.1.

- **Singly linked list**: each node points only to its successor. The last node's link is empty, written `null` in Java. The list can be walked only from front to back.
- **Doubly linked list**: each node points to both neighbors, so the list can be walked either way. The first node's previous link and the last node's next link are `null`.
- **Circularly linked list**: the last node points back to the first instead of to `null`, so the list is a ring with no empty link.

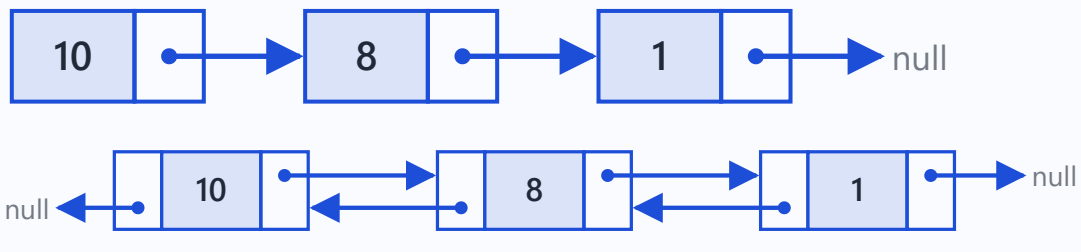




Figure 5.1: The three kinds of linked list, each holding 10, 8, 1. Top, singly linked: arrows go only forward, and the last node points to `null`. Middle, doubly linked: each pair of neighbors has a forward and a backward arrow, with `null` at both ends. Bottom, circular: the last node's link returns to the first.

All three keep elements in nodes that point to one another; they differ only in how many links a node holds and where the last link points. Those small differences can change costs dramatically: removing the last node means walking the whole singly linked list, but takes constant time in a doubly linked one.

5.1.2 The singly linked list

The simplest kind is the singly linked list, where each element holds a one-way link to the next. A **reference** is a value that tells where an object is in memory; in Java, every variable of a class type is a reference, holding the way to an object rather than the object itself.

The whole list is reached through one variable pointing to the first node, the **head**, written `head` in code. Starting there and following the links reaches every node. For an empty list, `head` is `null`.

Each node has two parts:

- **Data**: the value the node carries — a primitive such as an integer, or an object such as a student record.
- **Pointer**: a reference to the next node, `null` in the last node. In Java “pointer” and “reference” mean the same thing; there is no pointer arithmetic as in C.

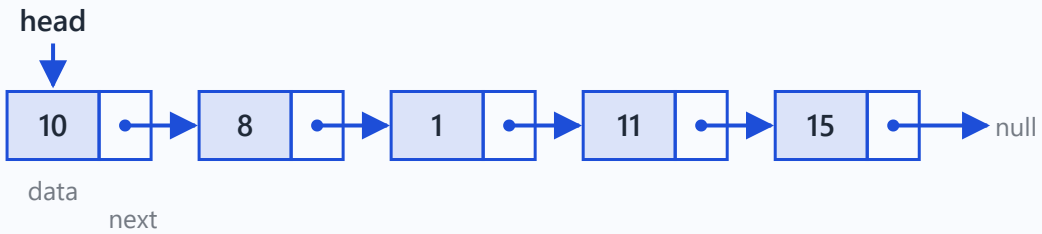


Figure 5.2: A singly linked list. The left part of each box holds the node's data (`data`), the right part the link to the next node (`next`). `head` points to the first node (10); the chain runs 10 → 8 → 1 → 11 → 15 and ends in `null`.

Two things in Figure 5.2 deserve attention. First, `head` is not a node, just a variable pointing to one. Second, nothing separate marks the end: the list ends at the node whose next field is `null`, and that field is how a program knows it has reached the last node.

> **Common mistake**
>
> Confusing `head` with the first element. `head` holds the way to the node, not the node. Redirect it elsewhere and the only path to the old first node may be lost; if nothing else refers to that node, it and everything after it are cut off from the program.

5.1.3 Where nodes live in memory

An array stores its elements contiguously, so finding element *i* takes one multiplication and one addition. A linked list allows no such computation. Each node is created separately with `new`, and Java's memory manager puts it wherever there is room; two nodes created in succession may land side by side or far apart.

A linked list is still **linear**: every element but the last has exactly one successor. A tree node may have several children; a list node has at most one next node. The references in the `next` fields, not memory positions, set the order.

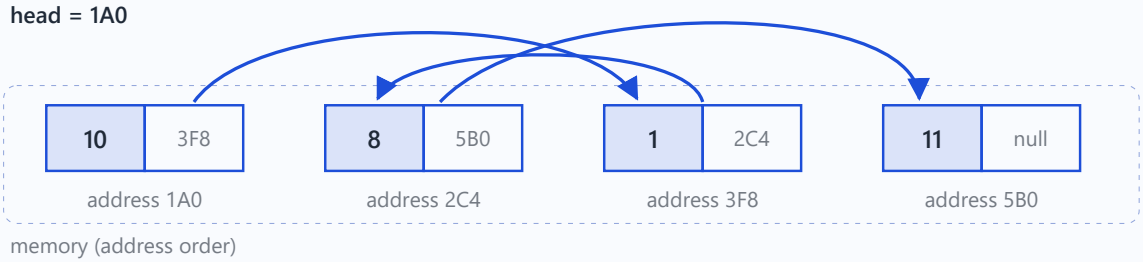


Figure 5.3: Nodes in memory, drawn in address order: 10 at 1A0, 8 at 2C4, 1 at 3F8, 11 at 5B0. Each node's right part holds the address of the next node. Starting at `head = 1A0` and following the arrows gives 10 → 1 → 8 → 11, not the address order 10, 8, 1, 11.

Figure 5.3 makes this concrete. The hexadecimal addresses are invented; Java programs never see real ones. The point is that the arrows ignore memory order: 10's node points to 3F8, the node of 1, which points back to 8 at 2C4, which points to 11 at 5B0, whose link is `null`. In memory 8 precedes 1; in the list it follows.

This has one good consequence and one bad one. Inserting in the middle shifts nothing; only a few links change. But you cannot jump to element *i*: with no address to compute, you start at the head and follow links one by one. These two facts determine the cost of every linked list operation.

5.1.4 The Node class

In Java a node is a small class. Here is one that holds an integer:

```
// Node class
class Node {
    int data;
    Node next;
    Node(int data) {
        this.data = data;
        this.next = null;
    }
}
```

The class has two fields: `data`, the integer the node carries, and `next`, whose type is again `Node`. A class with a field of its own type is a **self-referential structure**. The definition looks circular — a node containing a node containing a node — but `next` holds a reference to a node, not a node. References have a fixed size, so apart from its object header a `Node` occupies just one integer and one reference.

The constructor stores the value in `data` and sets `next` to `null`. Java initializes reference fields to `null` anyway; the line just states the intent. A new node is attached to nothing until someone stores its reference in another node's `next` field or in `head`.

A **generic** version stores data of any type:

```
// Node class
class Node<T> {
    T data;
    Node<T> next;
    Node(T data) {
        this.data = data;
        this.next = null;
    }
}
```

`T` is a type parameter: after `Node<String> n = new Node<>("Ann");`, `T` is `String` and the node stores text. For simplicity these notes use the integer version, but every algorithm works unchanged with the generic one, except that objects are compared with `equals` and `compareTo` instead of `==` and `<`.

Common mistake

Comparing values in a generic node with `==`, which asks whether two references point to the same object, not whether the values are equal. Two distinct `String` objects with the same text can give `false`. A search in a generic list should test `temp.data.equals(target)`.

5.1.5 Building a list by hand

The clearest way to see how a list is built is to create nodes and link them by hand. This code links four nodes in the order 10 → 1 → 8 → 11:

```
Node first = new Node(10);
Node second = new Node(1);
Node third = new Node(8);
Node fourth = new Node(11);
first.next = second;
second.next = third;
third.next = fourth;
```

The first four lines create four independent nodes; every `next` field is `null`, and each node is reachable only through its own variable (`first`, `second`, `third`, `fourth`). The last three lines build the links. `first.next = second` copies the reference in `second` into the `next` field of 10's node, which now points to 1's node.

Line	10.next	1.next	8.next	11.next	Chain from `first`
`first = new Node(10)`	null	—	—	—	10
`second = new Node(1)`	null	null	—	—	10
`third = new Node(8)`	null	null	null	—	10
`fourth = new Node(11)`	null	null	null	null	10
`first.next = second`	node(1)	null	null	null	10 → 1
`second.next = third`	node(1)	node(8)	null	null	10 → 1 → 8
`third.next = fourth`	node(1)	node(8)	node(11)	null	10 → 1 → 8 → 11

Table 5.1: Trace of the list-building code: each node's `next` field, and the chain reachable from `first`, after each line.



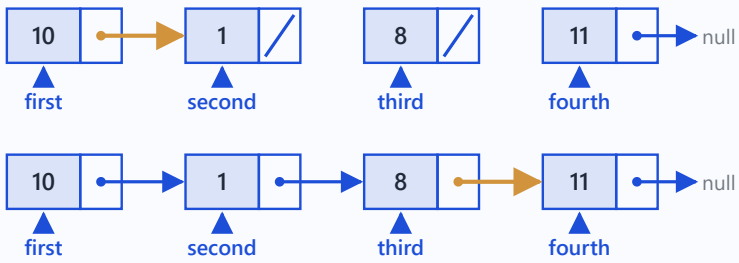


Figure 5.4: Building the list, top to bottom. (a) Four independent nodes with empty `next` fields (the slash means `null`). (b) `first.next = second`: the first link, 10 to 1, is in place (orange arrow). (c) All three links done: 10 → 1 → 8 → 11 → null.

The example shows a fact to keep in mind with linked lists: assignment copies the reference, never the object. After `first.next = second`, 1's node is reachable both through `second` and through `first.next` — one node, two paths. Once all links are set, the helper variables are unnecessary: the whole list is reachable from `first`, and `first.next.next.data` is 8.

Real programs call insertion methods instead of linking by hand, but each such method does exactly this inside: create a node and change a few reference fields.

5.1.6 Basic operations and their costs

The table summarizes the operations and their costs; each operation's own section justifies them. *n* is the number of nodes.

Operation	How it works	Time
Insert at front	The new node points to the old head and becomes the head.	O(1)
Insert at end	Walk to the last node; with a tail pointer, no walk is needed.	O(n) · O(1)
Remove from front	Move the head to the next node.	O(1)
Remove from end	The second-to-last node must be found.	O(n)
Search, print, length	Walk from front to back.	O(n)
Reverse, merge	One pass over the lists (merging uses a tail pointer).	O(n)

Table 5.2: Basic operations on a singly linked list.

One rule explains Table 5.2: *an operation that touches only nodes near the head is constant time; one that must walk into the list is linear.* The head is always known, so work there takes the same few steps however long the list is. The last node can be found only by following every link, so operations at the end are *O(n)* unless something like a tail pointer helps.

Accessing element *i* is *O(n)* too. An array reads `a[i]` in one step; a linked list starts at the head and follows *i* links — *n − 1* of them, in the worst case, for the last node.

Common mistake

Looping over a linked list by index: `for (int i = 0; i < n; i++) process(list.get(i));`. Each `get(i)` walks i steps from the head, for a total of $0 + 1 + \dots + (n - 1) = n(n - 1)/2$, or $O(n^2)$. The traversal pattern of the next section does the job in one $O(n)$ pass.

5.1.7 The list ADT and a Java class

A description of a data structure's operations, independent of their implementation, is an **abstract data type** (ADT). Goodrich, Tamassia, and Goldwasser define a small ADT for the singly linked list: size, emptiness test, first and last element, insertion at the front and the end, and removal from the front. We use the same names. The class skeleton:

```
public class SinglyLinkedList {
    private Node head = null, tail = null;
    private int size = 0;    // number of elements
    public int size() { return size; }
    public boolean isEmpty() { return size == 0; }
    public Integer first() {
        return isEmpty() ? null : head.data; }
      // addFirst, addLast, removeFirst …
}
```

Three fields: `head` points to the first node, `tail` to the last, and `size` counts the elements. All are `private`, so outside code must go through the methods, and keeping `head`, `tail`, and `size` true to the list's real state is the class's own job.

`size()` and `isEmpty()` only read `size`, so both are $O(1)$. `first()` returns `null` for an empty list and the front value otherwise; that is why it returns `Integer`, which can be `null`, rather than `int`, which cannot.

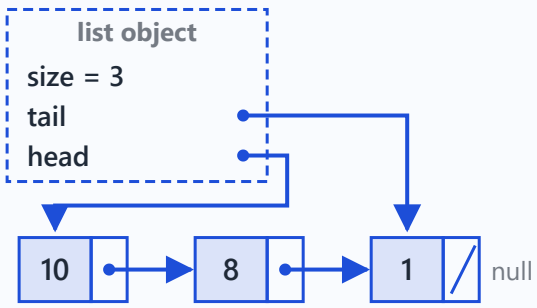


Figure 5.5: A `SinglyLinkedList` object. The dashed box is the list object, with fields `size = 3`, `head` (pointing to 10), and `tail` (pointing to 1). The nodes live outside it; the object reaches them only through the two references.

Goodrich–Tamassia	In these notes	Role	Time
`size()`, `isEmpty()`	`size()`, `isEmpty()`	the `size` field is updated by every operation	O(1)
`first()`, `last()`	`first()`, `isEmpty() ? null : tail.data`	return `null` on an empty list	O(1)
`addFirst`, `addLast`	`addFirst`, `addLast`	both constant time with a tail pointer; inserting into an empty list also sets `tail`	O(1)
`removeFirst()`	`removeFirst()`	returns the removed node	O(1)

Table 5.3: The textbook's ADT operations and how these notes implement them.

Every operation in Table 5.3 is constant time, provided every method that changes the list keeps `head`, `tail`, and `size` up to date together: insertions increment `size`, removals decrement it, the first node added to an empty list becomes both head and tail, and emptying a one-node list resets `tail` to `null`. Front insertion, end insertion with a tail pointer, and the removal methods do this. Only the walking version of end insertion ignores these fields, since it models the plain list without a tail pointer.

Common mistake

Keeping a `size` field but not updating it everywhere. If removals do `size--` but one insertion method skips `size++`, the counter drifts; after two insertions and a removal, `size()` may even return −1, and `isEmpty()`, which tests `size == 0`, goes wrong too. Either every modifying method updates the counter, or the length is counted each time.

5.1.8 Strengths and weaknesses

Both the strengths and the weaknesses come from nodes holding on to each other by links. The strengths:

- **It is easy to implement.** A node class and a head reference are all it takes.
- **Inserting and removing at the front are *O(1)*.** On an array both shift every element: *O(n)*.
- **Inserting in the middle shifts nothing.** The new node slips in by changing two links.
- **Its size is dynamic.** No capacity is reserved; each insertion does one `new`. The list never fills up, and nothing is resized or copied.

The weaknesses come from the same structure:

- **Search is slow.** Membership is decided only by checking nodes one by one: *O(n)*. Even a sorted list rules out binary search, since there is no jumping to the middle node.
- **There is no random access.** Reaching element *i* means walking from the head.

- **It uses extra memory.** Each node carries a reference besides its data — proportionally a heavy cost for small data: a single `int` also needs a reference and an object header.
- **It cannot be walked backward.** To find a node's predecessor, you start over from the head.

The *O(1)* cost of inserting or removing in the middle comes with a condition: the position — more precisely, the node before it — must *already be known*. Given only an index or a value, you first walk there, in *O(n)*. Claiming linked list insertion is *O(1)* without counting the walk is misleading.

> **Common mistake**
>
> Concluding that, since middle insertion is *O(1)* in a linked list and *O(n)* in an array, the linked list is always faster. Finding the middle is *O(n)* in a linked list too. The real gain comes from many insertions or removals in a row at one place: the position is found once and each operation after that is constant time, while an array shifts again every time.

5.2 Singly linked lists: traversal and insertion

Most linked list operations combine two moves: walking the list from front to back, and changing a few links in the right order. We start with traversal — length and search are small variations of it — and then cover insertion at the front, at the end, and at a given position. Most examples use 10 → 1 → 8 → 11.

5.2.1 Traversing and printing

Visiting every node of a list in order is called **traversal**. Printing is the simplest example: at each node, print its data.

```
void printList() {
    Node temp = head;
    System.out.print("Linked List: ");
    while (temp != null) {
        System.out.print(temp.data + " -> ");
        temp = temp.next;
    }
    System.out.println("null");
}
```

The method uses a local variable `temp`, first pointed at the node `head` refers to. While `temp` refers to a node, the loop prints its data and moves on with `temp = temp.next`. After the last node, `temp` takes that node's `next` value, `null`, and the loop ends; a final `null` marks the end of the list.

Note that `head` never changes. Only `temp` moves, and it vanishes when the method returns, leaving the list as it was.

Step	`temp`	Condition `temp != null`	Printed in this step
`temp = head`	node(10)	—	`Linked List:`
iteration 1	node(10) → node(1)	true	`10 ->`
iteration 2	node(1) → node(8)	true	`1 ->`
iteration 3	node(8) → node(11)	true	`8 ->`
iteration 4	node(11) → null	true	`11 ->`
loop ends	null	false	`null`

Table 5.4: Trace of `printList()` on the list 10 → 1 → 8 → 11.

The output is `Linked List: 10 -> 1 -> 8 -> 11 -> null`. The body runs four times and the condition is tested five times. For *n* nodes the loop runs *n* times with constant work each, so the time is *O(n)* and the extra memory *O(1)*.

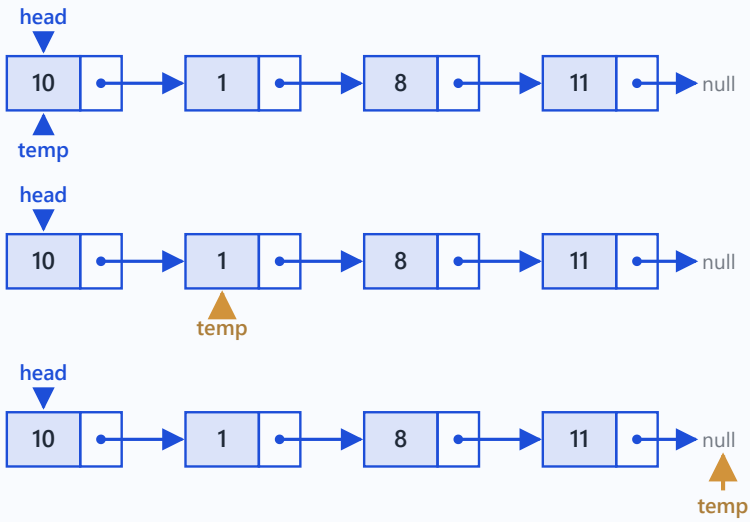


Figure 5.6: Three moments in the traversal. (a) `temp = head`: both point to 10. (b) After the first iteration `temp` is at 1; `head` stays put. (c) In the last iteration `temp` takes 11's `next`, `null`, and the loop ends.

Common mistake

Advancing `head` instead of a local variable: `while (head != null) { …; head = head.next; }`. Afterward `head` is `null`, the only path to the nodes is gone, and the list is effectively deleted; a second print shows only `null`. Always traverse with a copy, such as `Node temp = head;`.

5.2.2 The same pattern: length and search

The print method's skeleton recurs constantly. Call it the **traversal pattern**: start with `temp = head`; while `temp != null`, process the node; advance with `temp = temp.next`. Only the loop body changes. To find the length, the body increments a counter:

```
int listLength() {
    int count = 0;
    Node temp = head;
    while (temp != null) {
        count++;
        temp = temp.next;
    }
    return count;
}
```

The counter starts at zero and grows by one per node. An empty list skips the loop and returns 0 with no special case. The time is *O(n)*. If the length is needed often, the list keeps a `size` field, updated on every insertion and removal, and answers in *O(1)* — at the price that every modifying method must maintain it.

Search uses the same pattern; the body compares the node's data with the target:

```
int search(int target) {
    Node temp = head;
    int position = 0;
    while (temp != null) {
        if (temp.data == target)
            return position;
        position++;
        temp = temp.next;
    }
    return -1;   // not found
}
```

The counter `position` starts at 0 and follows `temp` node by node. When the target is found, its position is returned at once. If the loop ends without a match, the method returns −1, a safe “not found” because it is never a valid index.

The cost depends on where the target is: at the head, one comparison (best case *O(1)*); at the end or absent, all *n* nodes (worst case *O(n)*); at a random place, about *n/2* on average, still *O(n)*.

5.2.3 Code trace: search(8)

We search the list 10 → 1 → 8 → 11 for 8, which sits in the third node.

Example 5.1

1. **Start.** `temp = head`, that is, node(10); `position = 0`.
2. **Iteration 1.** `temp` is not `null`, so the loop is entered. `10 == 8` is false. `position` becomes 1 and `temp` moves to node(1).
3. **Iteration 2.** `1 == 8` is false. `position` becomes 2 and `temp` moves to node(8).
4. **Iteration 3.** `8 == 8` is true. The method executes `return position` and returns 2.

Three nodes, three comparisons. `search(5)` on the same list would check all four nodes, end the loop when `temp` became `null`, and return −1.

Iteration	`temp`	`position`	`temp.data == 8`	Result
1	node(10)	0	false	advance
2	node(1)	1	false	advance
3	node(8)	2	true	`return 2`

Table 5.5: Trace of `search(8)`. Each row shows the values just before that iteration's comparison.

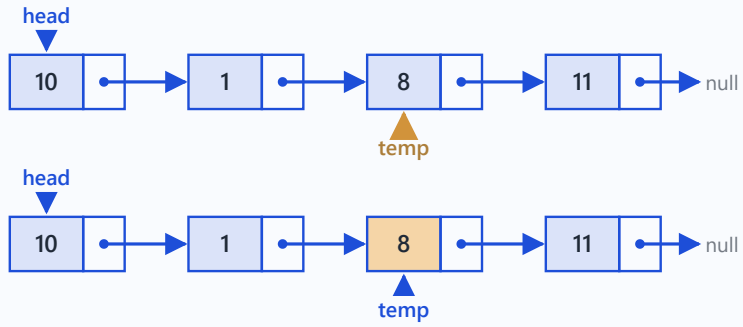


Figure 5.7: The last two moments of `search(8)`. (a) With `position` at 2, `temp` moves to node(8). (b) The comparison succeeds; 8 is at position 2.

Common mistake

Putting `position++` before the `if`. The counter is then one ahead at the comparison, and 8 is reported at 3. Increment it *after* checking the node, on the way to the next one.

5.2.4 Inserting at the front

Adding an element at the front is one of the fastest things a linked list can do:

```
void addFirst(int data) {
    Node newNode = new Node(data);
    newNode.next = head;
    head = newNode;
    if (tail == null) tail = head;
    size++;
}
```

The first line creates the node, with `next` set to `null`. The second copies `head` into the new node's `next`, so it points to the current first node. The third points `head` at the new node; the list now starts there, and the old nodes are reached through it.

The last two lines maintain the other fields. A node added to an empty list is also the last node, and since `tail` is still `null`, `if (tail == null) tail = head;` points it there. Otherwise `tail` is unaffected. `size++` counts the new element.

The order is the essence of the operation: attach the new node to the old list first, then move the head. It works for an empty list too — `head` is `null`, so the new node's `next` gets `null` and a one-node list results; the only extra step is setting `tail`.

Call	Line	`newNode`	`head`	`tail`	`size`	List
`addFirst(11)`	`new Node(11)`	11 → null	null	null	0	empty
	`newNode.next = head`	11 → null	null	null	0	empty
	`head = newNode`	11	node(11)	null	0	11 → null
	`tail = head`; `size++`	11	node(11)	node(11)	1	11 → null
`addFirst(8)`	`new Node(8)`	8 → null	node(11)	node(11)	1	11 → null
	`newNode.next = head`	8 → 11	node(11)	node(11)	1	11 → null
	`head = newNode`	8	node(8)	node(11)	1	8 → 11 → null
	`tail` unchanged; `size++`	8	node(8)	node(11)	2	8 → 11 → null
`addFirst(1)`	`new Node(1)`	1 → null	node(8)	node(11)	2	8 → 11 → null
	`newNode.next = head`	1 → 8	node(8)	node(11)	2	8 → 11 → null
	`head = newNode`	1	node(1)	node(11)	2	1 → 8 → 11 → null
	`tail` unchanged; `size++`	1	node(1)	node(11)	3	1 → 8 → 11 → null

Table 5.6: Trace of `addFirst(11)`, `addFirst(8)`, `addFirst(1)` on an empty list.

Table 5.6 shows something else: the elements went in as 11, 8, 1, but the list reads 1 → 8 → 11. Repeated front insertion reverses the order, just like a stack, where the newest element is in front.

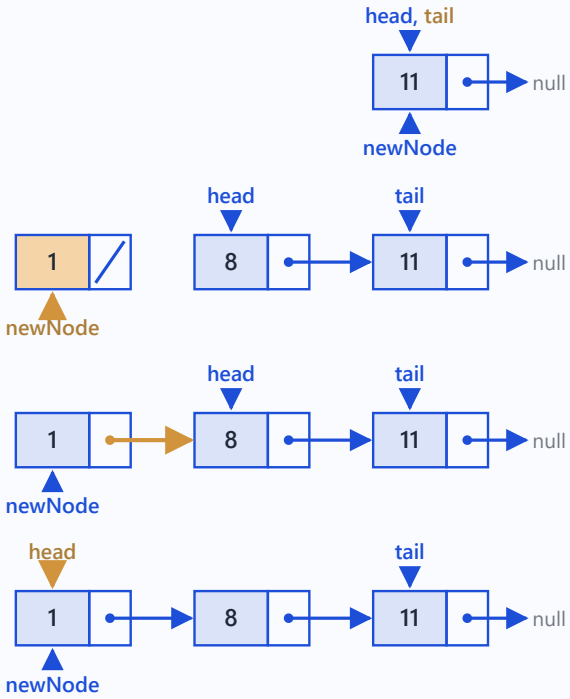


Figure 5.8: Front insertion in four moments. (a) After `addFirst(11)` on an empty list, 11 is head and tail. (b) `addFirst(1)`: node 1 is created, attached to nothing; `tail` is at 11. (c) `newNode.next = head`: 1 points to the old head, 8. (d) `head = newNode`: the list is 1 → 8 → 11 → null.

Whatever the length, front insertion is one object creation, two assignments, a comparison, and an increment. No node is visited and nothing shifts: *O(1)* time and *O(1)* extra memory. On an array, every element shifts right: *O(n)*.

Common mistake

Reversing the lines: `head = newNode;`, then `newNode.next = head;`. By then `head` is the new node, so the node ends up pointing to itself. The only reference to the old list is lost, and a traversal never ends, returning to the same node forever.

5.2.5 Inserting at the end

With only `head`, appending costs more: the new node must hang off the last node, which we first have to find:

```
void addLast(int data) {
    Node newNode = new Node(data);
    if (head == null) {
        head = newNode;
        return;
    }
    Node temp = head;
    while (temp.next != null)
        temp = temp.next;
    temp.next = newNode;
}
```

An empty list has nothing to walk; the new node becomes the head. Otherwise `temp` starts at the head and advances while `temp.next != null`, that is, while it has a *successor*, so it stops on the last node, and the final line links the new node there.

The two conditions differ in an important way. `temp != null` carries `temp` past the last node to `null`, leaving nothing to link from; `temp.next != null` stops one step earlier. Whenever the work is at the last node, use the second.

Call	Branch	Nodes visited by `temp`	Link set	List
`addLast(10)`	`head == null`	—	`head = node(10)`	10 → null
`addLast(1)`	walk	10 (next is null, stop)	10.next = node(1)	10 → 1 → null
`addLast(8)`	walk	10 → 1 (next is null, stop)	1.next = node(8)	10 → 1 → 8 → null

Table 5.7: Trace of `addLast(10)`, `addLast(1)`, `addLast(8)` on an empty list.

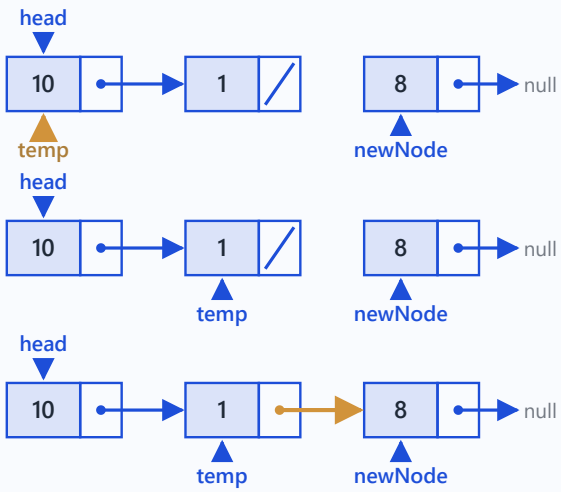


Figure 5.9: `addLast(8)` in three moments. (a) `temp = head`: the search for the last node starts at the head. (b) `temp` is at node(1), whose `next` is `null`: the loop ends. (c) `temp.next = newNode`: the list is 10 → 1 → 8 → null.

The walk costs $n - 1$ iterations: $O(n)$. Fine for one call, but n appends to an empty list take $0 + 1 + \ldots + (n - 1) = n(n - 1)/2$ steps, $O(n^2)$ — about 500,000 for 1,000 elements. A tail pointer removes this cost.

5.2.6 Inserting at the end with a tail pointer

The fix: keep the last node in a field too, the **tail pointer**, written `tail`. No walk is needed:

```
class SinglyLinkedList {
    Node head, tail;  int size;
    void addLast(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = tail = newNode;
        } else {
            tail.next = newNode;
            tail = newNode;
        }
        size++;
    }
}
```

In an empty list the new node is both head and tail: `head = tail = newNode` (Java assigns right to left, `tail` first). Otherwise `tail.next = newNode` links it after the old last node and `tail = newNode` moves the pointer. Either way, `size++` follows.

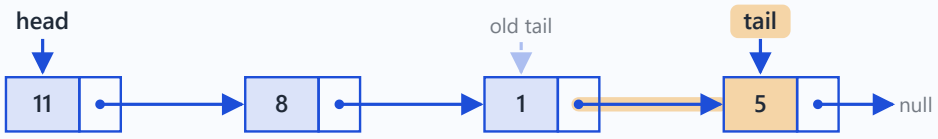


Figure 5.10: A list with head and tail pointers. Appending 5 to 11 → 8 → 1: the old tail (1) points to the new node (highlighted link), and `tail` is now 5. No node was visited.

Call	Line	`head`	`tail`	`size`	List
`addLast(10)`	`head = tail = newNode`	node(10)	node(10)	0	10 → null
	`size++`	node(10)	node(10)	1	10 → null
`addLast(1)`	`tail.next = newNode`	node(10)	node(10)	1	10 → 1 → null
	`tail = newNode`	node(10)	node(1)	1	10 → 1 → null
	`size++`	node(10)	node(1)	2	10 → 1 → null
`addLast(8)`	`tail.next = newNode`	node(10)	node(1)	2	10 → 1 → 8 → null
	`tail = newNode`	node(10)	node(8)	2	10 → 1 → 8 → null
	`size++`	node(10)	node(8)	3	10 → 1 → 8 → null

Table 5.8: Trace of `addLast(10)`, `addLast(1)`, `addLast(8)` with `tail`.

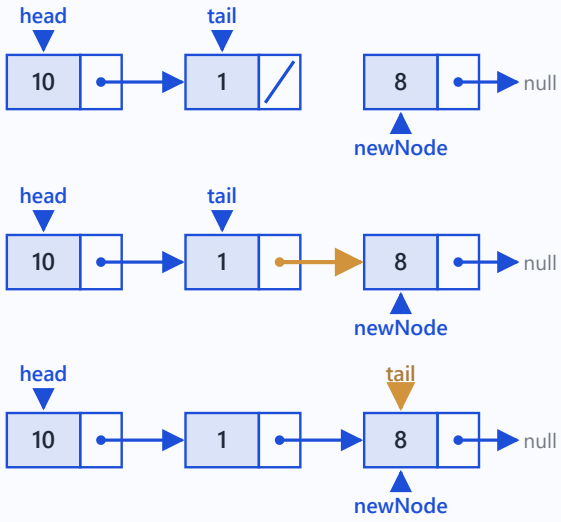


Figure 5.11: `addLast(8)` with `tail`. (a) The last node, node(1), is already known. (b) `tail.next = newNode`: the link 1 → 8 is set. (c) `tail = newNode`: the tail pointer moves to 8.

Each call in Table 5.8 makes at most two link assignments and one increment, regardless of length. Appending is now *O(1)*, and *n* appends take *O(n)*.

The tail pointer has two costs. First, maintenance: every operation that changes the list must keep `tail` right — front insertion into an empty list sets it, removing the last node moves it back, emptying the list nulls it. Miss one and `tail` points to a removed node or to `null`, so the next append goes astray or throws. Second, it doesn't speed up everything: removing from the end stays *O(n)*, because the link to change belongs to the second-to-last node, and there is no stepping back from `tail`.

Common mistake

Writing only `head = newNode;` in the empty-list branch. The first `addLast(5)` looks fine, but `tail` stays `null`, and the second, `addLast(7)`, takes the `else` branch, where `tail.next` throws a `NullPointerException`. The same happens if the first node comes from an `addFirst` that ignores `tail`.

5.2.7 Inserting at a given position

Positions start at 0: `addAt(data, 0)` inserts at the front, `addAt(data, 2)` makes the new element third.

```
void addAt(int data, int index) {   // index starts at 0
    if (index == 0) { addFirst(data); return; }
    Node temp = head;
    for (int i = 0; i < index - 1 && temp != null; i++)
        temp = temp.next;
    if (temp == null) {          // list shorter than index
        System.out.println("Position not found.");
        return;
    }
    Node newNode = new Node(data);
    newNode.next = temp.next;   // (1)
    temp.next = newNode;           // (2)
    if (temp == tail) tail = newNode;   // inserted at the end
    size++;
}
```

The method has three parts. The first line delegates index 0 to `addFirst`, since only front insertion changes `head`, which the rest of the code never touches.

The second part walks to the node *before* the target position, at *index − 1*, since its link is the one that changes. The loop advances `temp` *index − 1* steps; the `temp != null` test stops it if the list runs out, in which case the method prints a warning and returns, leaving the list untouched.

The third part links. Line (1) points the new node at `temp`'s current successor; line (2) points `temp` at the new node. The order — (1), then (2) — is critical. Finally, if the node went in at the end (`temp == tail`), `tail` moves to it, and `size` grows.

Example 5.2

On 10 → 1 → 8 → 11, `addAt(5, 2)` should put 5 third, between 1 and 8.

1. **Special case.** The index is not 0, so this is not a front insertion.
2. **Walk.** `temp = head` (node 10); the loop runs *index − 1 = 1* time. `i = 0`: `0 < 1`, so `temp` moves to node(1). `i = 1`: `1 < 1` fails; the loop ends.
3. **Check.** `temp` is node(1), not `null`; the position is valid.
4. **New node.** `new Node(5)`. 1 still points to 8; the list is unchanged.
5. **(1)** `newNode.next = temp.next`: 5 points to 8. The list is still unchanged, since nothing in it points to 5.
6. **(2)** `temp.next = newNode`: 1 points to 5, and the list is 10 → 1 → 5 → 8 → 11 → null.

Two links changed in all; 8 and 11 never moved in memory.

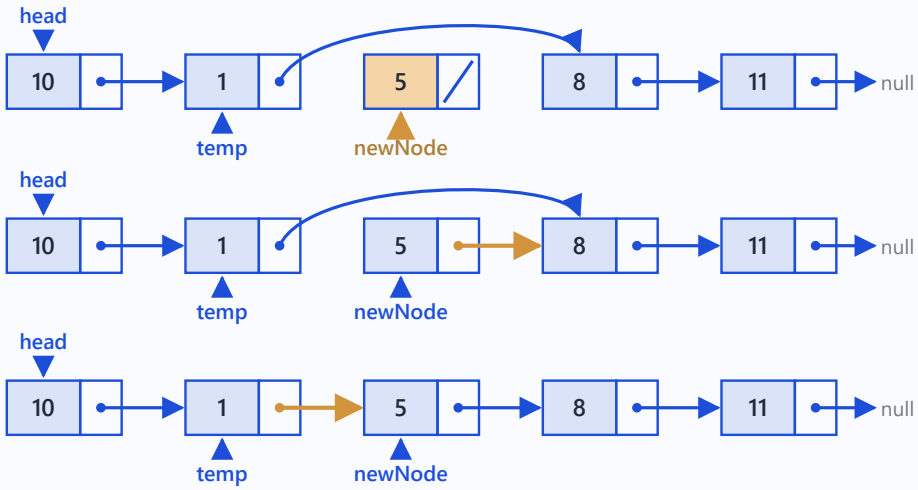


Figure 5.12: The linking steps of Example 5.2. (a) The new node 5 is created (8 and 11 are drawn shifted right; nothing moves in memory). (b) Line (1): 5 points to 8. (c) Line (2): 1 points to 5.

Reverse the order and (2) makes `temp.next` equal to 5; then (1) writes 5 into 5's own `next`. The node points to itself, the only reference to 8 is lost, 8 and 11 drop out, and a traversal loops 10 → 1 → 5 → 5 → ... forever.

Walking takes *index − 1* steps, *O(index)*, or *O(n)* near the end; linking is *O(1)*. If the index equals the length, `temp` stops at the last node and the node is appended; if larger, `temp` reaches `null` and "Position not found." is printed. On 10 → 1 → 8, `addAt(99, 3)` appends 99, while `addAt(99, 5)` changes nothing.

Common mistake

Writing the loop as `i < index`. `temp` then stops at the insertion point, not before it, and the node lands one place too far: `addAt(5, 2)` puts 5 after 8. For insertion and deletion, walk to the node *before* the one you operate on; in a singly linked list only the predecessor's link can change.

5.3 Singly linked lists: deletion and other operations

Deletion reverses insertion: a node leaves the chain when the link to it changes. Java needs no explicit freeing; once nothing refers to an object, the **garbage collector** (GC) reclaims it. So deletion's real job is to make the node unreachable without losing the rest of the list. The second half of the section covers classic problems: reversal, the middle element, copying, removing duplicates, merging, and sorting.

5.3.1 Removing from the front

Removing the first node is as cheap as front insertion. The method returns the removed node so the caller can use its value.

```
Node removeFirst() {
    if (head == null)
        return null;
    Node temp = head;
    head = head.next;  if (head == null) tail = null;
    temp.next = null;  size--;
    return temp;
}
```

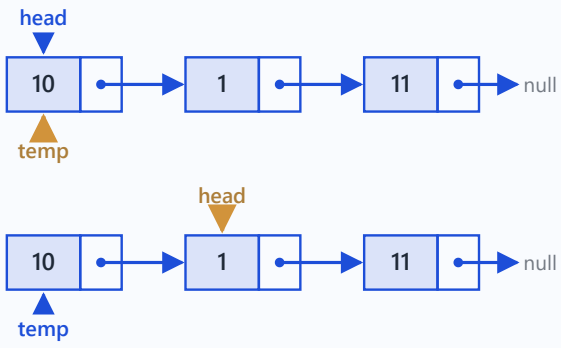
The first test catches the empty list and returns `null`; without it, `head.next` would throw a `NullPointerException`.

Otherwise the doomed node is saved in `temp` and `head = head.next` moves the head on; only `temp` still reaches the old first node. If the list is now empty, `tail` is nulled too, or it would keep pointing at a removed node. Then the node's `next` is cleared, `size` drops, and the node is returned.

`temp.next = null` isn't needed for correctness; it stops the returned node from pointing into the list, so nobody can reach the list through it by accident.

Call	`temp`	New `head`	`tail`	Returned	List
1st	node(10)	node(1)	node(11)	10	1 → 11 → null
2nd	node(1)	node(11)	node(11)	1	11 → null
3rd	node(11)	null	null	11	empty
4th	—	null	null	`null`	empty

Table 5.9: Four calls to `removeFirst()` on the list 10 → 1 → 11.



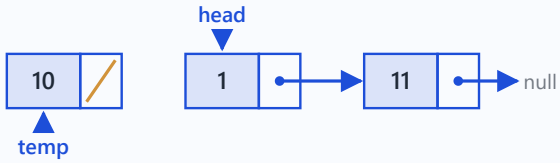


Figure 5.13: The first call, step by step. (a) `temp = head`: the node to be removed (10) is saved so it isn't lost. (b) `head = head.next`: the head moves to 1; 10 is unreachable from the list. (c) `temp.next = null`: 10 is fully detached.

Each call makes one comparison and a few assignments, independent of length: *O(1)*. If nobody keeps the returned node, the garbage collector eventually reclaims it.

5.3.2 Removing from the end

Removing the last node is expensive here: the link to change is in the second-to-last node, reachable only by walking from the head.

```
void removeLast() {
    if (head == null) return;
    if (head.next == null) {  // single node
        head = tail = null;  size--;
        return;
    }
    Node temp = head;
    while (temp.next.next != null)
        temp = temp.next;
    temp.next = null;  tail = temp;  size--;
}
```

Two special cases come first: an empty list needs nothing, and a one-node list (`head.next == null`), having no second-to-last node, is emptied by nulling `head` and `tail`.

In general, the loop tests `temp.next.next != null` — "is there a node two steps ahead?" When there isn't, `temp`'s successor is the last node, so `temp` is second to last; `temp.next = null` drops the last node and `tail` moves to `temp`.

Step	`temp`	`temp.next.next`	Decision
start	node(10)	node(8)	not null, advance
advance 1	node(1)	node(11)	not null, advance
advance 2	node(8)	null	stop: `temp` is second to last
unlink	`temp.next = null`, `tail = temp`: the list is 10 → 1 → 8 → null, `tail` = node(8)		

Table 5.10: Trace of `removeLast()` on the list 10 → 1 → 8 → 11.

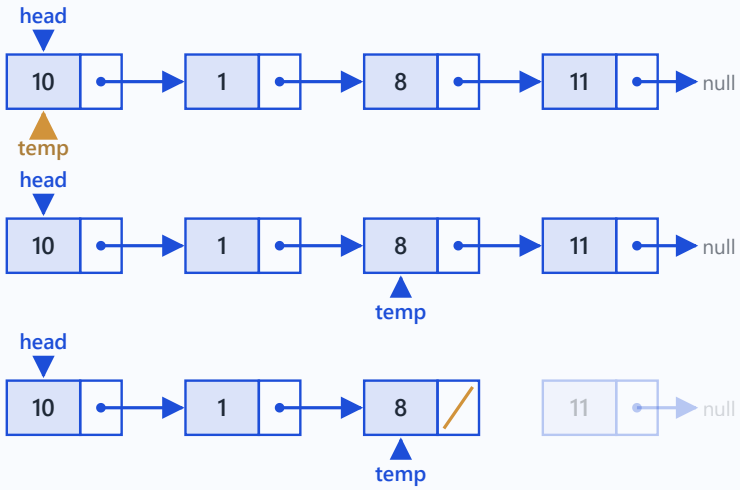


Figure 5.14: Removing from the end. (a) `temp = head`: the goal is the second-to-last node. (b) `temp` is at node(8); the link two steps ahead is `null`, so the loop ends. (c) `temp.next = null`: 11 leaves the list.

The loop runs $n - 2$ times: $O(n)$. A tail pointer doesn't help: deletion needs the node before `tail`, and a singly linked list cannot step back. This limitation is the main reason doubly linked lists exist.

Common mistake

Skipping the single-node check. If the list holds only 7, `temp.next` is `null`, and `temp.next.next` reads through it: `NullPointerException`. In a chain like `a.b.c`, every intermediate reference must be non-null.

5.3.3 Removing a given element

Removing by value means finding the element, then unlinking it through its predecessor. So two pointers walk together: `temp` on the node being checked, `prev` one behind.

```
void remove(int key) {
    if (head == null) return;
    if (head.data == key) {
        head = head.next;  size--;
        if (head == null) tail = null;  return;
    }
    Node prev = head;
    Node temp = head.next;
    while (temp != null) {
        if (temp.data == key) {
            prev.next = temp.next;
            if (temp == tail) tail = prev;  size--;
            return;
        }
        prev = temp;
        temp = temp.next;
    }
    System.out.println("Not found.");
}
```

The method handles three cases in turn.

1. **Empty list:** nothing to remove; return at once.
2. **Target at the head:** the head has no predecessor, so it is handled separately. `head` moves on and `size` drops; if the list is now empty, `tail` is nulled too.
3. **General case:** start with `prev = head`, `temp = head.next`. On a match, `prev.next = temp.next` skips the node (and `tail` moves to `prev` if it was the tail); otherwise both pointers advance. No match prints “Not found.”

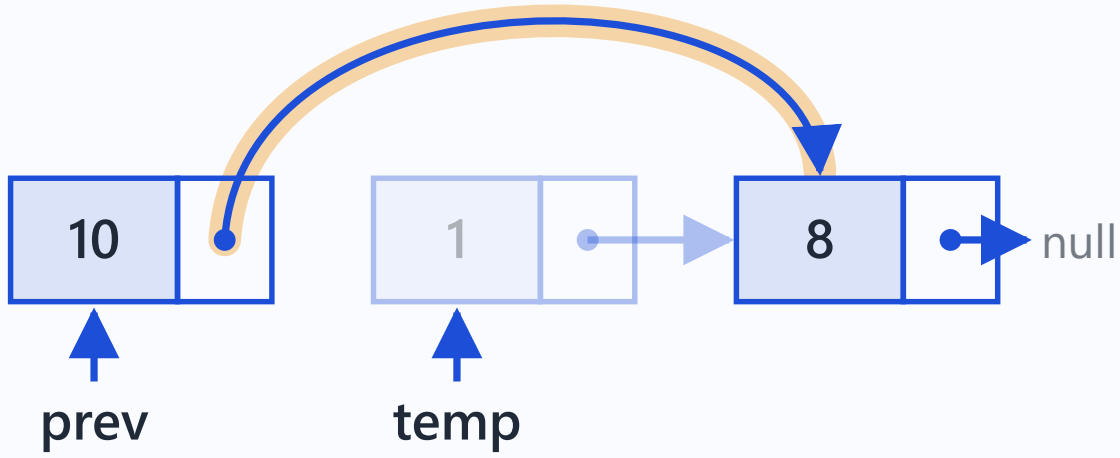


Figure 5.15: Deletion is a bypass: `prev.next = temp.next` points 10's link straight at 8 (highlighted). The removed node 1, faded, still points to 8, but nothing in the list leads to it.

Example 5.3

`remove(8)` on the list 10 → 1 → 8 → 11.

1. **Special cases.** The list isn't empty and the head, 10, isn't 8; both `if` s are passed.
2. **Start.** `prev` = node(10), `temp` = node(1).
3. **Iteration 1.** `1 == 8` is false. `prev` moves to node(1), `temp` to node(8).
4. **Iteration 2.** `8 == 8` is true. `prev.next = temp.next` : 1 now points to 11. `temp` isn't the tail, so `tail` stays. `size` drops and the method returns.

The result is 10 → 1 → 11 → null. The removed node 8 still points to 11, but nothing in the list reaches it.

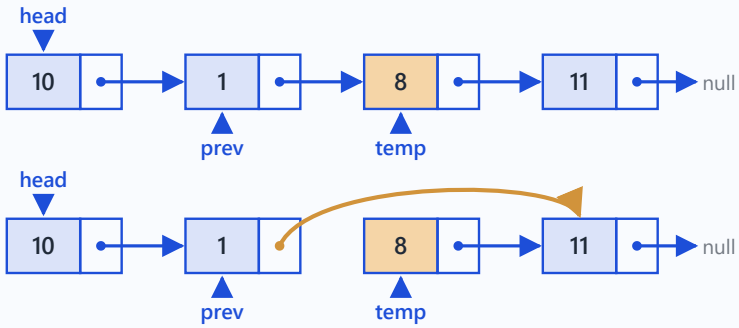


Figure 5.16: The last two steps of Example 5.3. (a) `temp` has found 8, with `prev` one behind at 1. (b) `prev.next = temp.next` : 1 points to 11, and 8 is bypassed.

The cost is the search: up to *n* nodes, *O(n)*, plus one *O(1)* assignment to unlink. Only the first match is removed: on 4 → 8 → 4 → 6, `remove(4)` deletes the leading 4 and leaves the second.

Common mistake

Writing `temp = temp.next;` on finding the node and believing it removed. That changes only the local variable; no link in the list is touched. The only way to remove a node is to change the link that points to it — the *predecessor's* `next` field.

5.3.4 Reversing a list

Reversal is a perennial interview favorite: 10 → 8 → 1 → 11 must become 11 → 1 → 8 → 10. A stack or a new list would work but waste memory; better to create no nodes and just turn the links around.

```
void reverse() {
    tail = head;                // the old head becomes the new tail
    Node prev = null;
    Node current = head;
    Node next = null;
    while (current != null) {
        next = current.next;  // 1) save the successor
        current.next = prev;   // 2) reverse the link
        prev = current;          // 3) prev advances
        current = next;          // 4) current advances
    }
    head = prev;
}
```

We walk the list once and turn each node's `next` link back toward its predecessor, using three pointers:

- `current` : the node whose link is being reversed.
- `prev` : where `current` 's new link will point; the head of the already reversed part.
- `next` : `current` 's successor, saved before the link changes — afterward the only way to the rest of the list.

Each iteration: (1) save the successor; (2) point `current` 's link back at `prev` ; (3) advance `prev` ; (4) advance `current` to the saved `next` . At the end `current` is `null` and `prev` holds the old last node, the new head: `head = prev` . The first line made the old head the tail.

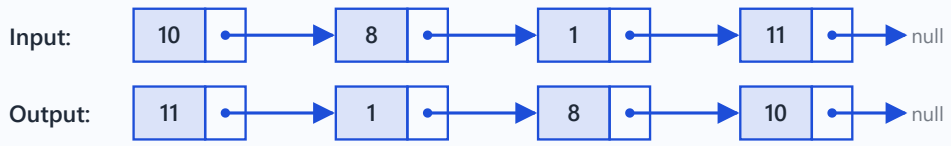


Figure 5.17: Input and output of reversal. Top, 10 → 8 → 1 → 11 → null; bottom, the same nodes as 11 → 1 → 8 → 10 → null. No node was created; only the links changed direction.

Iteration	`next`	Link reversed	`prev`	`current`	Reversed part
start	null	—	null	node(10)	empty
1	node(8)	10 → null	node(10)	node(8)	10 → null
2	node(1)	8 → 10	node(8)	node(1)	8 → 10 → null
3	node(11)	1 → 8	node(1)	node(11)	1 → 8 → 10 → null
4	null	11 → 1	node(11)	null	11 → 1 → 8 → 10 → null

Table 5.11: Trace of `reverse()` on 10 → 8 → 1 → 11, showing the state at the end of each iteration; “Reversed part” is the chain starting at `prev` .

After each iteration the list has two pieces: the reversed part from `prev` and the untouched part from `current`. Each iteration moves the first untouched node to the front of the reversed part — front insertion into a new list, reusing the same nodes.

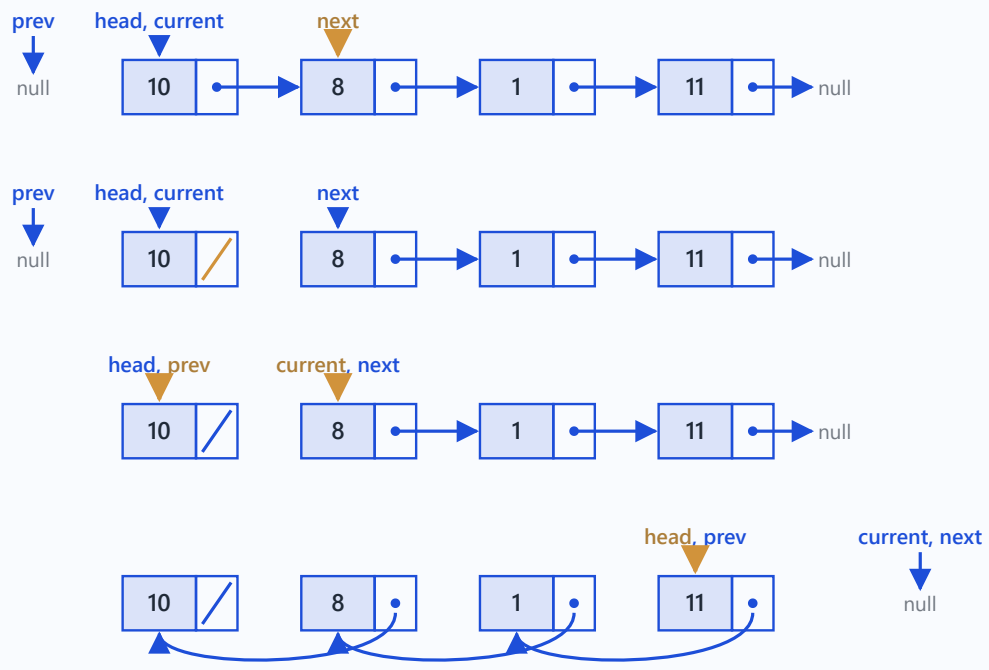


Figure 5.18: The first iteration's three steps and the result. (a) `next = current.next` : 8 is saved. (b) `current.next = prev` : 10's link turns to `null` ; only `next` still reaches 8. (c) `prev` and `current` advance. (d) After four iterations, `head = prev` : 11 → 1 → 8 → 10 → null.

One pass with four assignments per node: *O(n)* time; three pointers, no new nodes: *O(1)* extra memory. An empty list skips the loop, and a one-node list comes out unchanged, with no special checks.

Common mistake

Omitting line (1) or putting it after (2). Once `current.next = prev` runs, the link to the rest is gone; an unsaved successor is unreachable and reversal stops after one node. Forgetting `head = prev` leaves `head` on the old first node, now at the *end*, so the list looks like a single node.

5.3.5 The middle element: slow and fast pointers

The obvious way to find the middle takes two passes: count the length *n*, then walk *n/2* steps. The **slow and fast pointers** technique needs one. Both start at the head; each iteration the slow one moves one node and the fast one two. When the fast pointer reaches the end, the slow one is halfway.

```
int findMiddle() {
    if (head == null)
        throw new IllegalStateException("List is empty");
    Node slow = head, fast = head;
    while (fast != null && fast.next != null) {
        slow = slow.next;          // 1 step
        fast = fast.next.next;  // 2 steps
    }
    return slow.data;
}
```

Two steps need both `fast` and its successor, hence `fast != null && fast.next != null`. With an odd count, `fast` stops on the last node; with an even count it overshoots to `null`. Either way `slow` ends in the middle. An empty list has no middle, so the method throws.

Example 5.4

Find the middle of 10 → 8 → 1 → 11 → 5 (five nodes).

1. **Start.** `slow` and `fast` are at node(10).
2. **Iteration 1.** `fast` (10) and its successor (8) exist. `slow` moves to 8, `fast` two steps to 1.
3. **Iteration 2.** `fast` (1) and its successor (11) exist. `slow` moves to 1, `fast` to 5.
4. **End.** `fast.next` is `null`; the loop ends and the method returns `slow.data`, 1.

In two iterations `slow` took 2 steps and `fast` 4. 1 is the third of five nodes — exactly the middle.

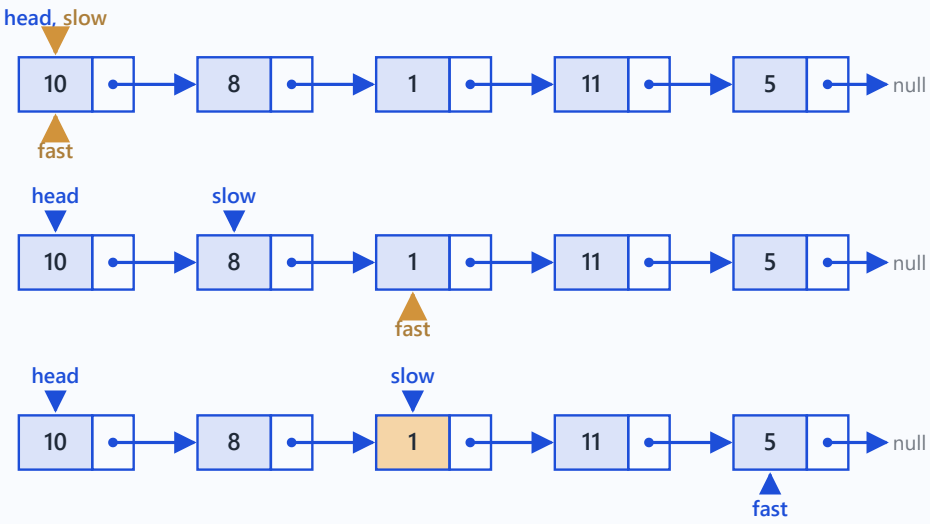


Figure 5.19: Slow and fast pointers. (a) Both start at the head. (b) After the first iteration `slow` is at 8 and `fast` at 1. (c) When `fast` reaches the last node, `slow` is at the middle, 1.

An even-length list has two middles; in 10 → 8 → 1 → 11 they are 8 and 1. This code returns the *second*: after two iterations `fast` is `null` and `slow` is at 1. For the first middle, change the condition to `fast.next != null && fast.next.next != null`.

About *n/2* iterations: *O(n)* time, *O(1)* extra memory. The step count matches the two-pass method; the gain is one traversal without knowing the length. The same idea detects cycles (Section 5.5.3).

Common mistake

Writing the condition as just `fast.next != null`. On an even-length list `fast` becomes `null`, and the next test reads `null.next`: `NullPointerException`. The `fast != null` part is needed, and it must come first, because `&&` skips the second operand when the first is false.

5.3.6 Copying a list

Copying a list means building a separate list of *new nodes* with the same values, so that changes to the copy leave the original alone.

```
SinglyLinkedList copy() {
    SinglyLinkedList newList =
        new SinglyLinkedList();
    Node temp = head;
    while (temp != null) {
        newList.addLast(temp.data);
        temp = temp.next;
    }
    return newList;
}
```

The method creates an empty list, traverses the original, and appends each value to the copy with `addLast`. Since `addLast` does a `new Node` every time, the copy's nodes are entirely separate from the original's. This is a **deep copy**.

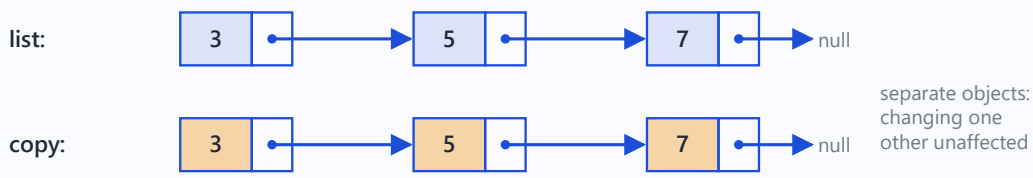


Figure 5.20: A deep copy. Top, the original 3 → 5 → 7; bottom, its copy. The values match, but the nodes are different objects, so changing one list doesn't affect the other.

The tempting shortcut `newList.head = head` creates no nodes, only a second reference to the same ones — a **shallow copy**; changing a value in one changes the other. A real copy of 10 → 1 → 8 also reads 10 → 1 → 8, but `newList.head == head` is `false`.

With a tail pointer each append is *O(1)* and copying is *O(n)*; with only `head`, the appends total $n(n-1)/2$ steps, $O(n^2)$. The n new nodes take *O(n)* memory.

> **Common mistake**
>
> Copying with `addFirst`. It is cheap but reverses the order: 10 → 1 → 8 becomes 8 → 1 → 10. Without a tail pointer, to keep the order, hold the copy's last node in a local variable and link each new node directly to it.

5.3.7 Removing duplicates

A value may occur more than once. The goal is to keep only the first occurrence of each: 3 → 5 → 3 → 7 → 5 becomes 3 → 5 → 7.

```
void removeDuplicates() {
    SinglyLinkedList seen =
        new SinglyLinkedList();
    Node prev = null, temp = head;
    while (temp != null) {
        int v = temp.data;
        if (seen.contains(v)) {
            prev.next =
                temp.next;   // skip
            if (temp == tail) tail = prev;  size--;
        } else {
            seen.addFirst(v);
            prev = temp;
        }
        temp = temp.next;
    }
}
```

Values already met go into a second list, `seen`, queried with `contains(v)`, which returns `search(v) != -1`. A seen value marks a duplicate, skipped with `prev.next = temp.next`; `prev` stays, because its new successor is still unchecked. A new value goes into `seen` and `prev` advances. Either way `temp` moves on.

`prev` starts as `null`, yet `prev.next` never fails: the first value can't have been seen, so the first node always takes the `else` branch.

Node	`contains(v)`	Action	`prev` (after)	`seen`	List
3 (1st)	false	`addFirst(3)`, `prev` advances	node(3)	3	3 → 5 → 3 → 7 → 5
5 (2nd)	false	`addFirst(5)`, `prev` advances	node(5)	5 → 3	3 → 5 → 3 → 7 → 5
3 (3rd)	true	5's link turns to 7; 3 is skipped	node(5)	5 → 3	3 → 5 → 7 → 5
7 (4th)	false	`addFirst(7)`, `prev` advances	node(7)	7 → 5 → 3	3 → 5 → 7 → 5
5 (5th)	true	7's link becomes `null`; the node was the tail, so `tail = prev`	node(7)	7 → 5 → 3	3 → 5 → 7

Table 5.12: Trace of `removeDuplicates()` on 3 → 5 → 3 → 7 → 5.

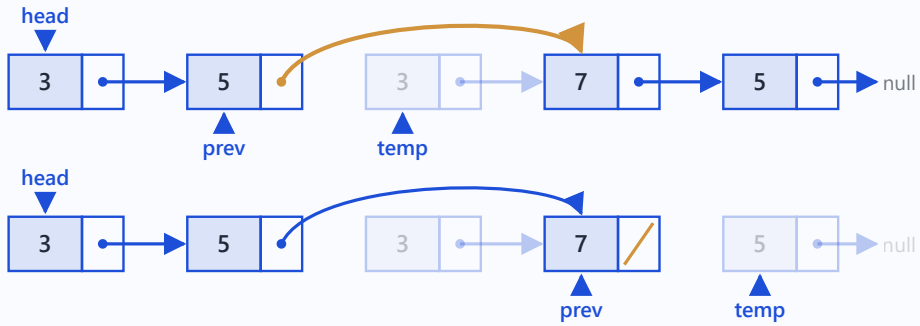


Figure 5.21: Two bypasses. (a) The 3 in the third node is a duplicate: 5's link goes straight to 7, and `prev` stays at 5. (b) The 5 in the last node is a duplicate: 7's link turns to `null`. The result is 3 → 5 → 7 → null.

The `contains` calls dominate: one search of up to n values per node, $O(n^2)$ in all. A **hash set** makes each lookup $O(1)$ on average and the total $O(n)$. Extra memory is the number of distinct values, at most $O(n)$.

Using `addFirst` for `seen` is deliberate: order doesn't matter there, only membership, and front insertion, at $O(1)$, is the cheapest way in.

Common mistake

Advancing `prev` after skipping a duplicate, which moves it onto the removed node. With two duplicates in a row, as in 2 → 2 → 2, the second bypass changes the removed node's link and the third 2 stays in the list. When a node is removed, `prev` stays; only `temp` moves.

5.3.8 Merging two sorted lists

Merging two ascending lists yields one sorted list of all their elements: 1 → 4 → 7 and 2 → 3 → 8 give 1 → 2 → 3 → 4 → 7 → 8. It is also the core step of merge sort.

```
SinglyLinkedList merge(
        SinglyLinkedList l1,
        SinglyLinkedList l2) {
    SinglyLinkedList result =
        new SinglyLinkedList();
    Node p1 = l1.head, p2 = l2.head;
    while (p1 != null && p2 != null) {
        if (p1.data <= p2.data) {
            result.addLast(p1.data);
            p1 = p1.next;
        } else {
            result.addLast(p2.data);
            p2 = p2.next;
        }
    }
```

```
    // append the remaining elements
    while (p1 != null) {
        result.addLast(p1.data);
        p1 = p1.next;
    }
    while (p2 != null) {
        result.addLast(p2.data);
        p2 = p2.next;
    }
    return result;
}
```

`p1` and `p2` start at the two heads. While both lists have elements, the smaller current value is appended and only its pointer advances. On a tie, `<=` takes the first list's element, preserving the order of equal elements.

When one list runs out, the other's leftovers are sorted and no smaller than anything appended, so they go in without comparisons. At most one remainder loop runs.

Iteration	`p1`	`p2`	Comparison	Appended	`result`
1	1	2	$1 \le 2$ true	1	1
2	4	2	$4 \le 2$ false	2	$1 \to 2$
3	4	3	$4 \le 3$ false	3	$1 \to 2 \to 3$
4	4	8	$4 \le 8$ true	4	$1 \to 2 \to 3 \to 4$
5	7	8	$7 \le 8$ true	7	$1 \to 2 \to 3 \to 4 \to 7$
remainder	null	8	— (main loop done)	8	$1 \to 2 \to 3 \to 4 \to 7 \to 8$

Table 5.13: Merging $1 \to 4 \to 7$ with $2 \to 3 \to 8$.

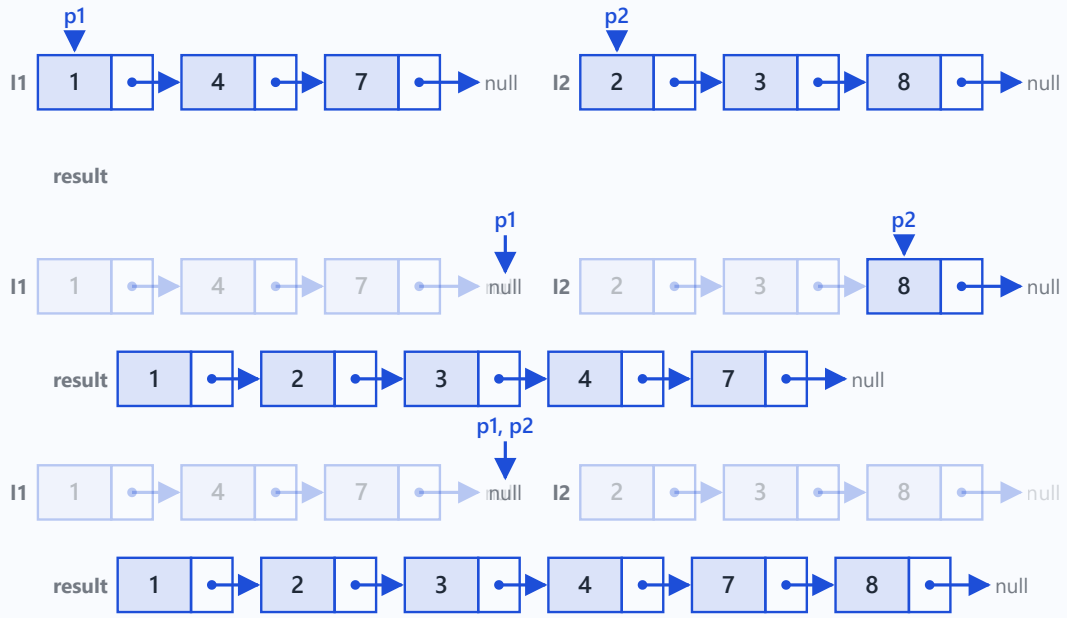


Figure 5.22: Merging. (a) `result` is empty; `p1` and `p2` are at the two heads. (b) After five comparisons `p1` reaches `null`; only 8 remains in the second list. (c) 8 is appended without comparison: 1 → 2 → 3 → 4 → 7 → 8.

Each of the $n + m$ elements is appended once, with at most $n + m - 1$ comparisons (5 here). With an $O(1)$ `addLast` the time is $O(n + m)$; appends that walk the result would make it $O((n + m)^2)$. The result's new nodes take $O(n + m)$ memory; relinking the original nodes instead needs $O(1)$ but destroys the inputs.

> **Common mistake**
>
> Forgetting the remainder loops. The main loop stops when one list is exhausted, so leftovers like the 8 here never arrive, and the result is 1 → 2 → 3 → 4 → 7.

5.3.9 Sorting a list

A list can also be sorted without moving nodes, by swapping the values in their data fields. A simple exchange sort:

```
void sort() {
    Node current = head;
    while (current != null) {
        Node other = current.next;
        while (other != null) {
            if (current.data > other.data) {
                int temp = current.data;
                current.data = other.data;
                other.data = temp;
            }
            other = other.next;
        }
        current = current.next;
    }
}
```

The outer loop walks `current` along the list; the inner loop visits every later node with `other`, **swapping** values through the local `temp` whenever `current`'s is larger. After each inner loop, `current` holds the smallest value from there on, and the outer loop moves on.

`current`	`other`	Comparison	Swap	List
node 1	node 2	4 > 1	yes	1 → 4 → 3 → 2
node 1	node 3	1 > 3	no	1 → 4 → 3 → 2
node 1	node 4	1 > 2	no	1 → 4 → 3 → 2
node 2	node 3	4 > 3	yes	1 → 3 → 4 → 2
node 2	node 4	3 > 2	yes	1 → 2 → 4 → 3
node 3	node 4	4 > 3	yes	1 → 2 → 3 → 4
node 4	null	—	—	1 → 2 → 3 → 4

Table 5.14: Trace of `sort()` on 4 → 1 → 3 → 2. Positions are numbered 1 to 4; the nodes stay in place and only the values change.

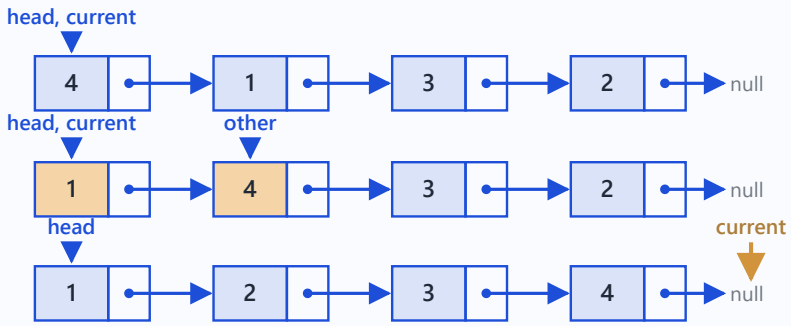


Figure 5.23: Sorting by swapping values. (a) Start: 4 → 1 → 3 → 2. (b) First comparison, 4 > 1: the values are swapped, giving 1 → 4 → 3 → 2. (c) Result: 1 → 2 → 3 → 4. No node moved and no link changed.

Table 5.14 has 3 + 2 + 1 = 6 comparisons. In general node k costs $n - k$, for $n(n - 1)/2$ in total whatever the initial order, sorted or not. Swaps depend on the input (4 here). Time $O(n^2)$, extra memory $O(1)$.

Merge sort suits linked lists better: split with slow and fast pointers, sort the halves, merge — $O(n \log n)$, and relinking needs no extra array. Index-hungry methods such as quicksort and heapsort fit lists poorly.

Common mistake

Trying to swap nodes by swapping two local variables: `Node t = current; current = other; other = t;` . No link changes; the pointers just trade names and the loops break. Real node swaps update both nodes' links and their predecessors'. Swapping values is far simpler.

5.4 Doubly linked lists

A singly linked list cannot step back from a node. That is why removing from the end costs $O(n)$, why removing an element needs two pointers, and why backward printing is impossible. A doubly linked list gives each node one more link.

5.4.1 Definition, node, and structure

Definition 5.2

A **doubly linked list** is a linked list in which every node holds references to both the previous and the next node. The first node's previous link and the last node's next link are `null` . The list is usually kept with both a head (`head`) and a tail (`tail`) reference and can be traversed in either direction.

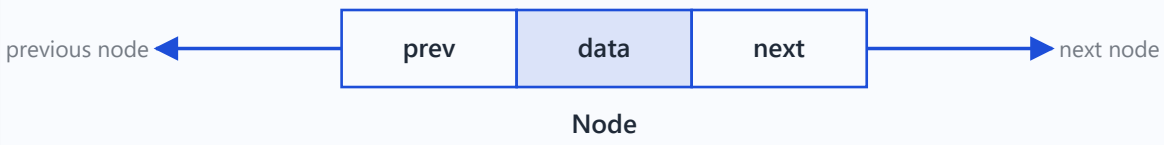


Figure 5.24: A doubly linked node has three fields: `prev` , pointing to the previous node; `data` , holding the value; and `next` , pointing to the next node.

The node class gains one field:

```
class Node {
    int data;
    Node prev;
    Node next;
    Node(int data) {
        this.data = data;
        this.prev = null;
        this.next = null;
    }
}
```

The list class keeps `Node head, tail;`. The gain: backward traversal, and *O(1)* removal of a known node and of the last node. The price: a second reference per node, a noticeable memory increase for a list of integers, and links in both directions to update on every change — roughly twice as many.

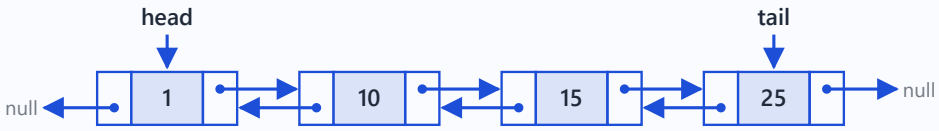


Figure 5.25: The doubly linked list 1 ⇄ 10 ⇄ 15 ⇄ 25. `head` points to 1, `tail` to 25. Each pair of neighbors has a forward and a backward link; the empty links at the two ends point to `null`.

Every operation follows these rules:

- Elements are reachable both forward and backward.
- Inserting a node updates pointers in both directions: the new node's two links and each neighbor's link toward it.
- Removing a node updates the pointers of both its predecessor and its successor.
- Search can go either way; for a target known to be near the end, start from the tail.

Correctness is one invariant: *for every node* `d`, *if* `d.next` *is not* `null` *then* `d.next.prev == d`, *and if* `d.prev` *is not* `null` *then* `d.prev.next == d`. Every forward link is matched by a backward one, and an operation is correct only if it preserves this.

5.4.2 Inserting at the front and the end

Front insertion adds one assignment to the singly linked version's two:

```
public void addFirst(int data) {
    Node newNode = new Node(data);
    if (head == null) {
        head = newNode;
        tail = newNode;
    } else {
        newNode.next = head;
        head.prev = newNode;
        head = newNode;
    }
}
```

An empty list gets the node as head and tail. Otherwise: the new node's `next` points to the old head (forward link); the old head's `prev`, so far `null`, points back to it; `head` moves to it. Its own `prev` stays `null`, as the first node's must.

Line	1.prev	1.next	10.prev	`head`
`new Node(1)`	null	null	null	node(10)
`newNode.next = head`	null	node(10)	null	node(10)
`head.prev = newNode`	null	node(10)	node(1)	node(10)
`head = newNode`	null	node(10)	node(1)	node(1)

Table 5.15: Trace of `addFirst(1)` on 10 ⇄ 15 ⇄ 25.

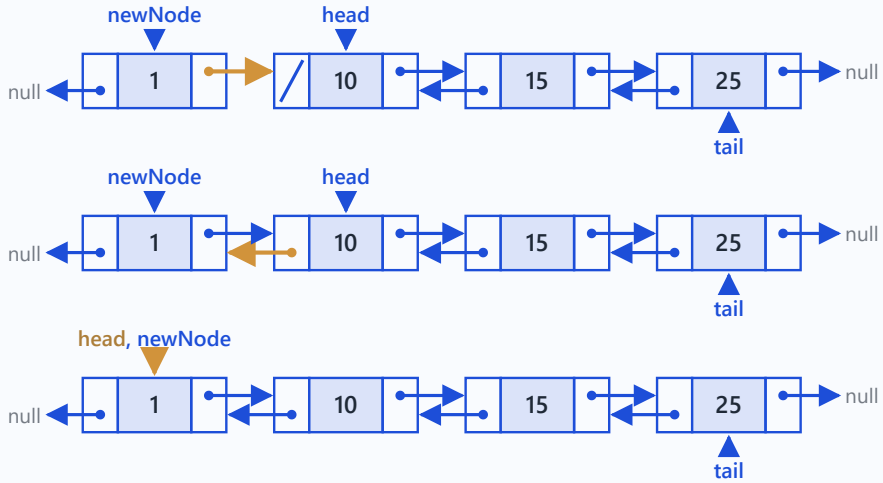


Figure 5.26: `addFirst(1)` on a doubly linked list. (a) `newNode.next = head`: forward link 1 → 10. (b) `head.prev = newNode`: backward link 10 → 1, replacing 10's `null`. (c) `head = newNode`: the list is 1 ⇄ 10 ⇄ 15 ⇄ 25.

Insertion at the end mirrors it:

```
public void addLast(int data) {
    Node newNode = new Node(data);
    if (head == null) {
        head = newNode;
        tail = newNode;
    } else {
        newNode.prev = tail;
        tail.next = newNode;
        tail = newNode;
    }
}
```

The new node's `prev` points to the old tail, the old tail's `next` to it, and `tail` moves to it — no traversal. `addLast(30)` on 1 ⇄ 10 ⇄ 15 ⇄ 25 gives 1 ⇄ 10 ⇄ 15 ⇄ 25 ⇄ 30, which reads 30, 25, 15, 10, 1 backward.

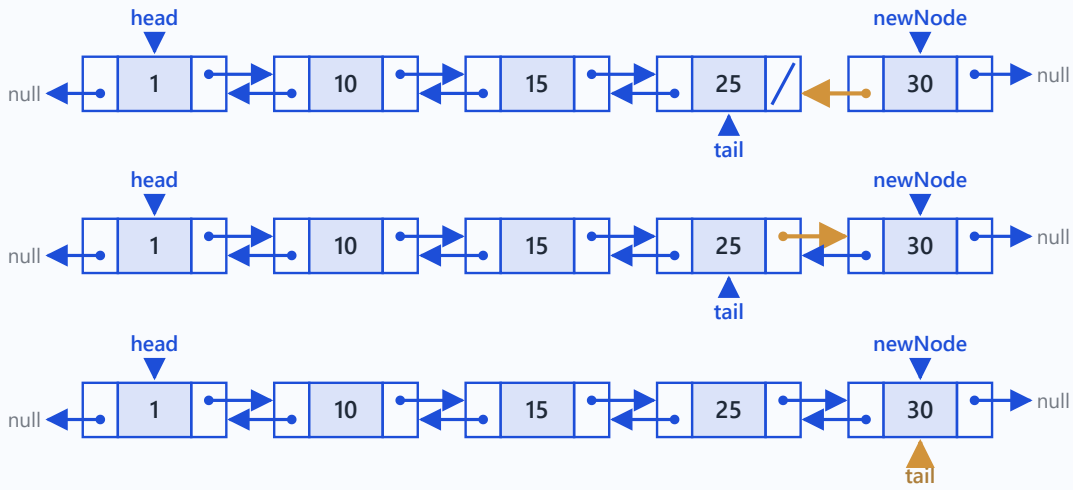


Figure 5.27: `addLast(30)` on a doubly linked list. (a) `newNode.prev = tail` : backward link 30 → 25. (b) `tail.next = newNode` : forward link 25 → 30. (c) `tail = newNode` : the tail pointer moves to 30.

Both are *O(1)*, at one assignment more than the singly linked versions — the price of cheap backward traversal and removal at the end.

Common mistake

Setting only the forward links. Without `head.prev = newNode` , forward printing looks right and hides the bug, but backward printing stops at 10, missing 1, and removals working back from 10 update the wrong node. Printing both ways after each change catches this early.

5.4.3 Inserting at a position

Again we walk to the node before the insertion point, but linking changes four links instead of two.

```
void addAt(int data, int index) {   // index starts at 0
    if (index == 0) { addFirst(data); return; }
    Node temp = head;
    for (int i = 0; i < index - 1 && temp != null; i++)
        temp = temp.next;
    if (temp == null) { System.out.println("Invalid position."); return; }
    if (temp == tail) { addLast(data); return; }   // also updates tail
    Node newNode = new Node(data);
    newNode.next = temp.next;     // (1)
    newNode.prev = temp;              // (2)
    temp.next.prev = newNode;      // (3)
    temp.next = newNode;             // (4)
}
```

The opening matches the singly linked version: index 0 goes to `addFirst`, `temp` walks to node *index − 1*, an invalid position prints a warning. New: if `temp` is the last node, `addLast` takes over, because line (3), `temp.next.prev`, would read through the last node's `null`; `addLast` also updates `tail`.

The four links are set in this order — the new node's own two first, leaving the list untouched, then the neighbors' links toward it:

1. `newNode.next = temp.next`: the new node points to its right neighbor.
2. `newNode.prev = temp`: the new node points to its left neighbor.
3. `temp.next.prev = newNode`: the right neighbor's backward link turns to the new node.
4. `temp.next = newNode`: the left neighbor's forward link turns to the new node.

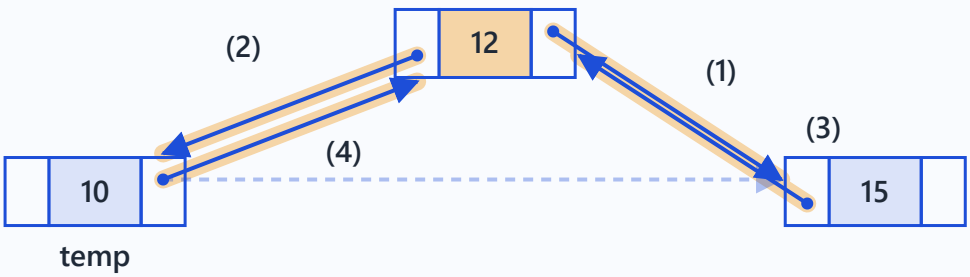


Figure 5.28: Inserting 12 between 10 and 15. The numbers are code lines: (1) 12 → 15, (2) 12 → 10, (3) 15's backward link to 12, (4) 10's forward link to 12. The dashed line is the direct link between 10 and 15 before the insertion.

Example 5.5

`addAt(12, 2)` on 1 ⇄ 10 ⇄ 15 ⇄ 25.

1. **Walk.** `temp` advances *index − 1 = 1* step to node(10), which is not the last node.
2. **(1)** 12.next = 15. No node in the list points to 12 yet.
3. **(2)** 12.prev = 10.
4. **(3)** `temp.next` is still 15; 15.prev becomes 12.
5. **(4)** 10.next becomes 12.

Forward the list reads 1, 10, 12, 15, 25; backward, 25, 15, 12, 10, 1. Four links changed.

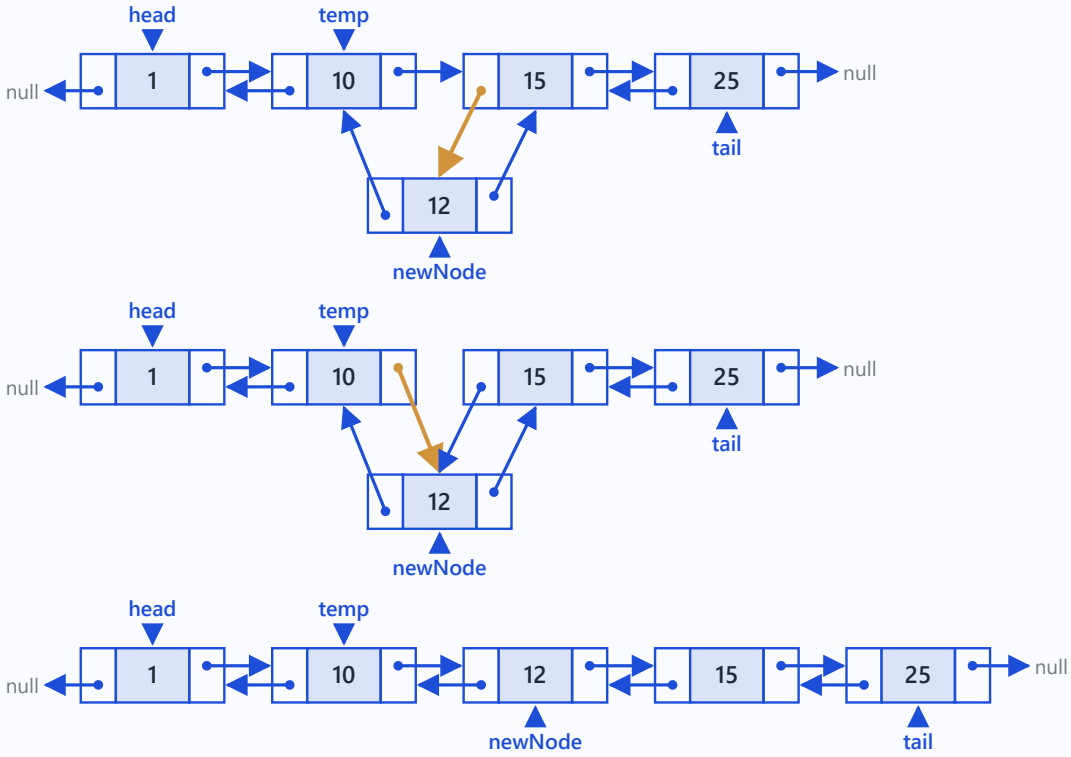


Figure 5.29: The last steps of Example 5.5. (a) (3): 15's backward link points to 12. (b) (4): 10's forward link points to 12. (c) The result, redrawn: 1 ⇄ 10 ⇄ 12 ⇄ 15 ⇄ 25.

Why must (3) precede (4)? Line (3) reaches 15 through `temp.next`. Run (4) first and `temp.next` is 12, so (3) sets 12.prev = 12, while 15 still points back to 10. Forward reading (1, 10, 12, 15, 25) hides the bug; backward reading (25, 15, 10, 1) skips 12. Rule: when you reach a node through a link, finish with that node before changing the link.

Linking is *O(1)*; the walk is *O(index)*, *O(n)* at worst. For an index in the second half, walking back from the tail takes at most *n/2* steps — still *O(n)*.

5.4.4 Removing from the front and the end

In a doubly linked list, removal at either end is constant time:

```
void removeFirst() {
    if (head == null) return;  // empty
    if (head == tail) {  // single node
        head = tail = null;
    } else {
        head = head.next;
        head.prev = null;
    }
}
```

```
void removeLast() {
    if (tail == null) return;  // empty
    if (head == tail) {  // single node
        head = tail = null;
    } else {
        tail = tail.prev;
        tail.next = null;
    }
}
```

One skeleton serves both. An empty list needs nothing; a one-node list (`head == tail`) is emptied by nulling both fields. Otherwise the front version moves `head` on and nulls its `prev` ; the end version moves `tail` back and nulls its `next` .

The speed comes from `tail = tail.prev` : one step to the second-to-last node, which a singly linked list finds only by walking the whole list.

Operation	Line	Effect	List
`removeFirst()` , 1 ⇄ 10 ⇄ 15	`head = head.next`	head moves to 10	(10 still points back to 1)
	`head.prev = null`	10's backward link is cleared	10 ⇄ 15
`removeLast()` , 1 ⇄ 10 ⇄ 15 ⇄ 25	`tail = tail.prev`	tail moves to 15	(15 still points forward to 25)
	`tail.next = null`	15's forward link is cleared	1 ⇄ 10 ⇄ 15

Table 5.16: Trace of removal at both ends of a doubly linked list.

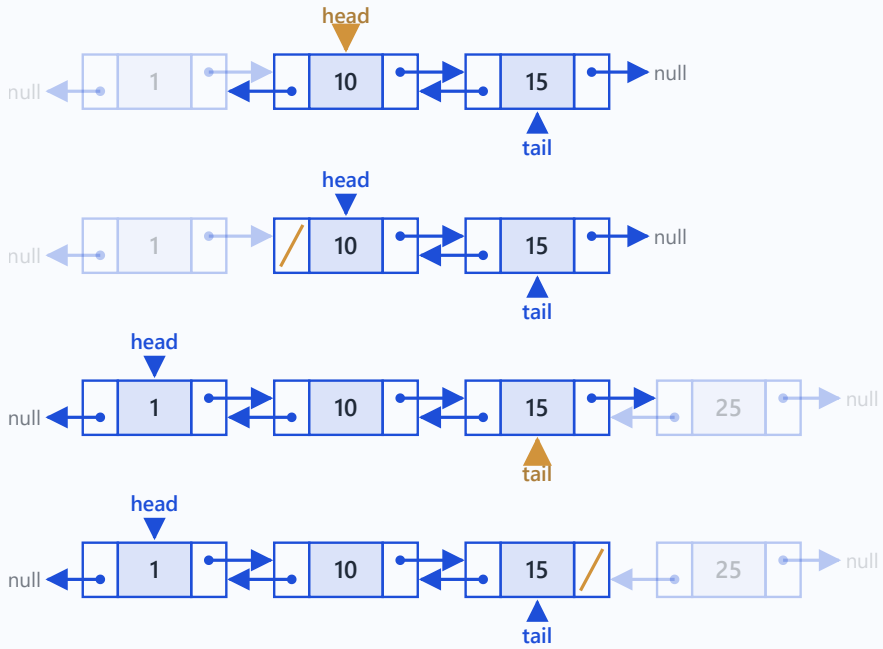


Figure 5.30: Removal at both ends. (a) `head = head.next` : the head moves to 10. (b) `head.prev = null` : 10's backward link is cleared too. (c) `tail = tail.prev` : one step reaches the second-to-last node. (d) `tail.next = null` : 25 leaves the list, giving 1 ⇄ 10 ⇄ 15.

Each method is *O(1)*. In the one-node case, `head` and `tail` must be nulled *together*; otherwise the list looks empty from one end and nonempty from the other, and the next insertion takes the wrong branch.

Common mistake

Skipping `head.prev = null` . Forward traversal still works, but the new head points back to the removed node, so backward printing prints the removed 1 too — and the lingering reference keeps the garbage collector from reclaiming it.

5.4.5 Removing a given element

The singly linked version needed `prev` and `temp` in step. Here one pointer suffices, since the predecessor is `temp.prev` .

```
void remove(int data) {
    Node temp = head;
    while (temp != null && temp.data != data)
        temp = temp.next;
    if (temp == null) return;  // not found
    if (temp.prev != null)
        temp.prev.next = temp.next;
    else head = temp.next;   // removing the head
    if (temp.next != null)
        temp.next.prev = temp.prev;
    else tail = temp.prev;    // removing the tail
}
```

The loop advances while `temp` is non-null and holds another value, ending on the target or at `null`, meaning the value is absent.

Two links bypass the node: the left neighbor's forward link goes to the right neighbor, and the right neighbor's backward link to the left one. At the ends, `head` (no left neighbor) or `tail` (no right neighbor) is updated instead; in a one-node list both become `null`.

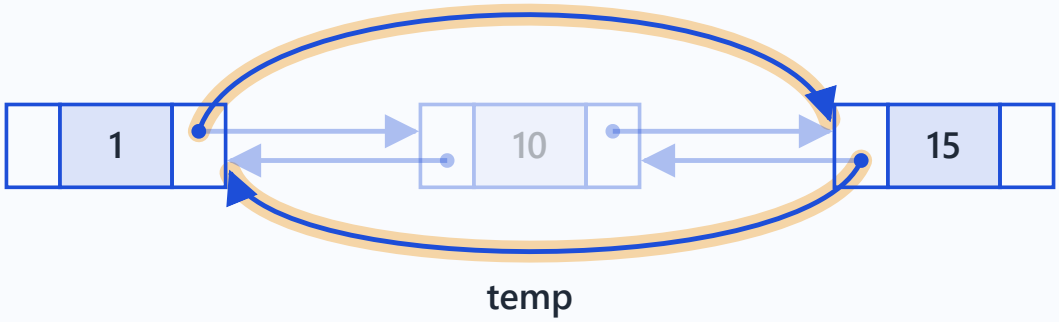


Figure 5.31: Removing from the middle. To remove 10, 1's forward link goes to 15 and 15's backward link to 1 (highlighted). The removed node is faded.

Step	`temp`	Check	Action
search	node(1)	1 ≠ 15	advance
search	node(10)	10 ≠ 15	advance
search	node(15)	15 == 15	loop ends
left	node(15)	`temp.prev` = 10, not null	10.next = 25
right	node(15)	`temp.next` = 25, not null	25.prev = 10

Table 5.17: Trace of `remove(15)` on 1 ⇄ 10 ⇄ 15 ⇄ 25.

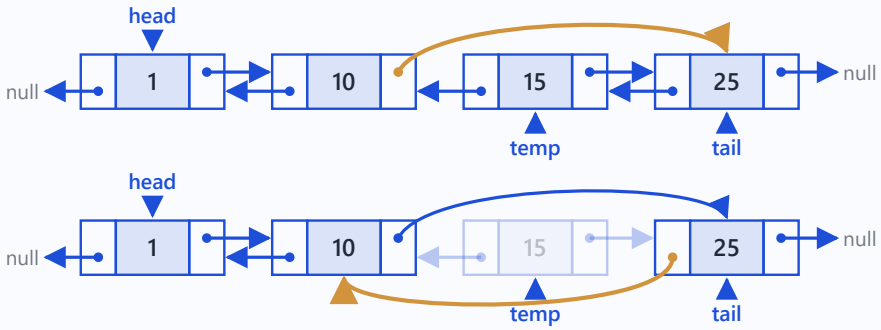


Figure 5.32: The linking steps of `remove(15)`. (a) `temp.prev.next = temp.next`: forward link 10 → 25; 15 is bypassed going forward. (b) `temp.next.prev = temp.prev`: backward link 25 → 10. The result is 1 ⇄ 10 ⇄ 25.

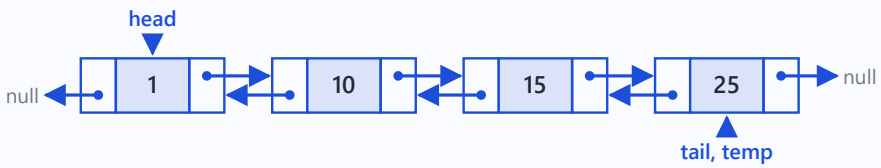
Search is *O(n)*, unlinking *O(1)*. The real gain comes when the node is already in hand: a program that keeps node references removes one in *O(1)* through its `prev` and `next`. A singly linked list still has to walk from the head to find the predecessor.

5.4.6 Printing backward

Backward printing mirrors forward printing: start at `tail` and follow the `prev` links.

```
void printBackward() {
    Node temp = tail;
    System.out.print("Backward: ");
    while (temp != null) {
        System.out.print(temp.data + " <-> ");
        temp = temp.prev;
    }
    System.out.println("null");
}
```

No walk is needed to reach the last node. Each iteration prints `temp`'s data and steps back with `temp = temp.prev`, ending past the first node, whose `prev` is `null`. For 1 ⇄ 10 ⇄ 15 ⇄ 25 the output is `Backward: 25 <-> 15 <-> 10 <-> 1 <-> null`.



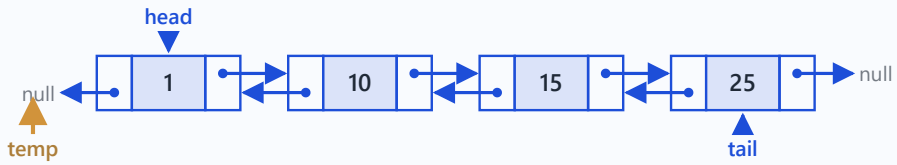


Figure 5.33: Printing backward. (a) `temp = tail`: the traversal starts at the last node. (b) After 1 is printed, `temp` takes 1's `prev`, `null`, and the loop ends.

Both directions are *O(n)*. A singly linked list can print backward by recursion or with a stack, but either costs *O(n)* extra memory (for recursion, the call stack); the doubly linked list needs *O(1)*.

5.4.7 Sentinel nodes

The same special cases recur in every doubly linked method: empty list? head? tail? They lengthen the code and breed bugs — miss one and you get a `NullPointerException` or a wrong `head` / `tail`. There is an elegant way to eliminate them.

Definition 5.3

A **sentinel** node is a fixed node placed at one end of a list that carries no real element. A doubly linked list with sentinels has two: a `header` and a `trailer` (Goodrich and Tamassia's terms). The real nodes always lie between them.

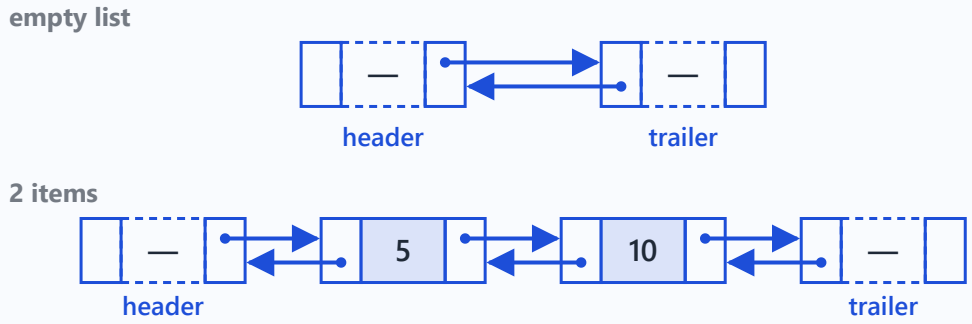


Figure 5.34: A doubly linked list with sentinels. Top, the empty list: just the two sentinels, pointing to each other. Bottom, a two-element list: 5 and 10 sit between `header` and `trailer`. The sentinels' data fields are shown as "—".

With sentinels, every real node *always* has a predecessor and a successor, so `null` checks and `head` / `tail` updates vanish. Even the empty list is ordinary: two sentinels pointing to each other, linked once at construction and never separated. Two small helpers do all insertions and removals:

```
void addBetween(int v, Node pred, Node succ) {
    Node newest = new Node(v);
    newest.prev = pred;
    newest.next = succ;
    pred.next = newest;
    succ.prev = newest;  size++;
}
void addFirst(int v) {
    addBetween(v, header, header.next); }
void remove(Node d) {
    d.prev.next = d.next;
    d.next.prev = d.prev;
    size--; }
```

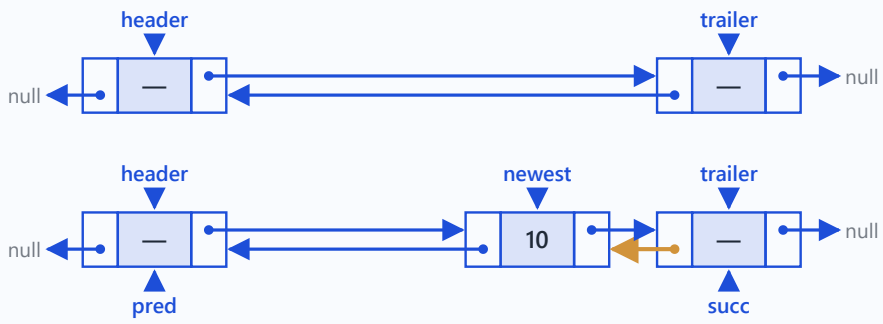
`addBetween(v, pred, succ)` inserts between two neighbors with four assignments. `addFirst` calls it with the header and the first real node; appending uses `addBetween(v, trailer.prev, trailer)`. `remove(d)` links `d`'s neighbors to each other. No method contains an `if`.

Example 5.6

On an empty list with sentinels we call `addFirst(10)` and `addFirst(5)`, then remove 10's node with `remove`.

1. **Start.** `header.next = trailer`, `trailer.prev = header`, `size = 0`.
2. `addFirst(10)`. This calls `addBetween(10, header, trailer)`, since in the empty list `header.next` is the trailer. Four assignments: 10.prev = header, 10.next = trailer, header.next = 10, trailer.prev = 10. `size = 1`.
3. `addFirst(5)`. `addBetween(5, header, node(10))`; the same four assignments put 5 between the header and 10. List: header ⇄ 5 ⇄ 10 ⇄ trailer; `size = 2`.
4. `remove(d)`, **d = node(10).** `d.prev.next = d.next`: 5 points straight to the trailer. `d.next.prev = d.prev`: the trailer points back to 5. `size = 1`.

10 was the last real node. Without sentinels, removing it would have required updating `tail`; here nothing special happened.



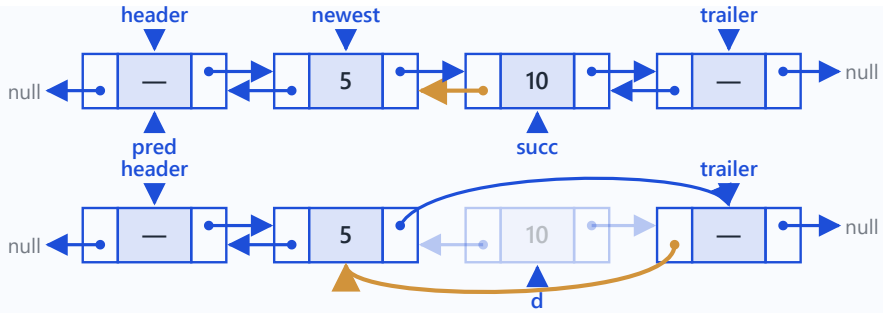


Figure 5.35: Four moments of Example 5.6. (a) The empty list: the two sentinels point to each other. (b) After `addFirst(10)`. (c) After `addFirst(5)`: 5 is between the header and 10. (d) `remove` has bypassed 10 in both directions.

Every insertion and removal is *O(1)*, and shorter code means fewer branches and fewer bugs. The price is two extra nodes, and traversal runs from `header.next` while `temp != trailer`, not `temp != null`. Java's `java.util.LinkedList` is also doubly linked, and many textbook implementations use sentinels.

Common mistake

Starting a traversal at `header` prints its meaningless data; running to `null` prints the trailer too. Sentinels are not data: start at the first real node and stop at the trailer.

5.4.8 Links in more directions: a linked structure from a matrix

Links can run in whatever directions a structure needs. A nice example turns a two-dimensional matrix into a linked structure whose nodes hold `right` (the cell to the right) and `down` (the cell below): `class Node { int data; Node right, down; }`, with constructor `Node(int data)`.

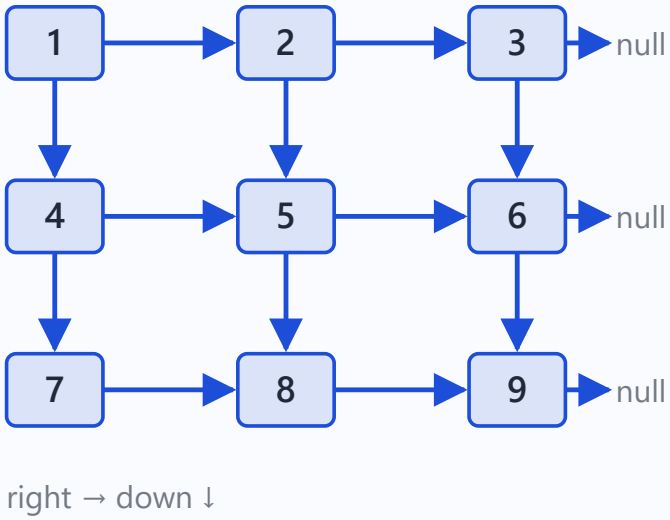


Figure 5.36: The linked structure of a 3×3 matrix. Each node points to the one on its right and the one below. The `right` field is `null` in the last column, and `down` is `null` in the last row. The whole structure is reached from the top-left node.

Two nested traversals from the top-left node print it row by row:

```
void display(Node start) {
    Node down = start;
    while (down != null) {       // rows
        Node right = down;
        while (right != null) {     // columns
            System.out.print(right.data + " ");
            right = right.right;
        }
        System.out.println();
        down = down.down;
    }
}
```

The outer loop follows `down` links from row to row, the inner loop `right` links along a row. For Figure 5.36 the output is `1 2 3`, `4 5 6`, `7 8 9` on three lines. Each node is visited once: *O(rows · columns)*.

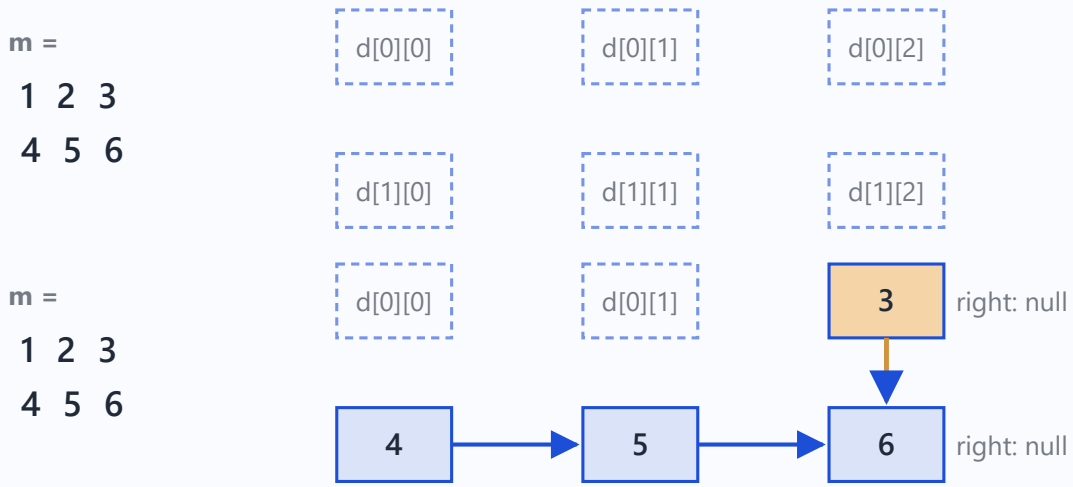
Building it is more interesting. Setting a node's `right` and `down` requires those neighbors to exist already, so creation starts at the bottom-right corner and runs backward:

```
Node build(int[][] m) {
    int r = m.length, c = m[0].length;
    Node[][] d = new Node[r][c];
    for (int i = r - 1; i >= 0; i--)
        for (int j = c - 1; j >= 0; j--) {
            Node n = new Node(m[i][j]);
            if (j + 1 < c) n.right = d[i][j+1];
            if (i + 1 < r) n.down = d[i+1][j];
            d[i][j] = n;
        }
    return d[0][0];
}
```

Nodes are stored in a helper array `d` at their matrix positions. Because `i` and `j` count down, `d[i][j+1]` and `d[i+1][j]` exist when cell `(i, j)` is built; in the last column and last row the corresponding link stays `null`. The top-left node is returned.

(i, j)	New node	`right`	`down`
(1, 2)	6	none (last column)	none (last row)
(1, 1)	5	6	none (last row)
(1, 0)	4	5	none (last row)
(0, 2)	3	none (last column)	6
(0, 1)	2	3	5
(0, 0)	1	2	4

Table 5.18: Trace of `build(m)` for the 2×3 matrix `{{1, 2, 3}, {4, 5, 6}}`.



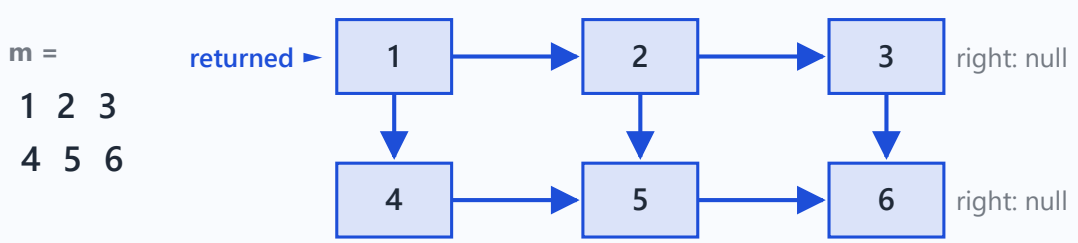


Figure 5.37: Three moments of `build(m)`. (a) The array `d` is empty. (b) The bottom row's three nodes and 3 have been created; 3's `down` link points to 6. (c) Six nodes for six cells; `d[0][0]` (1) is returned.

One node per cell: *O(rows · columns)* time and the same memory for `d`. An *r*-by-*c* matrix gets *r(c − 1)* `right` and *(r − 1)c* `down` links — 6 and 6 for 3×3.

Why the table `d`? The obvious recursion creates node `(i, j)` and calls `build(i, j+1)` for `right` and `build(i+1, j)` for `down`, reaching each cell by many routes and creating a node each time: (1, 1) is reached from (0, 1) and from (1, 0), yielding two nodes. Each cell gets one node per right-down path from the corner — 19 instead of 9 for 3×3, and exponentially worse as the matrix grows. Since the paths never meet at a shared node, the result isn't even a grid. The table creates each node once.

Common mistake

Running the loops from the top left: `for (int i = 0; i < r; i++)`. `d[i][j+1]` and `d[i+1][j]` are still `null` when node `(i, j)` is created, so all links stay `null` and nine disconnected nodes result. Create a node before linking to it.

5.5 Circularly linked lists

Some work goes round in turns. In a board game the turn passes from player to player, back to the first after the last. An operating system hands waiting programs short time slices in turn. A music player on “repeat” follows the last song with the first. Such lists have no end; the end is tied to the beginning.

5.5.1 Definition and class

Definition 5.4

A **circularly linked list** is a linked list whose last node points to the first node instead of to `null`. The nodes form a ring; each still points to its successor, and no link in the list is `null`.

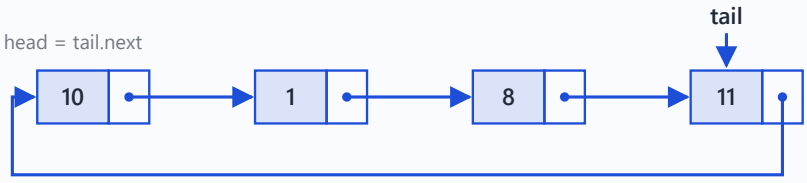


Figure 5.38: The circular list 10 → 1 → 8 → 11. The last node (11) points back to the first (10). The list is kept through `tail` alone; the head is `tail.next`.

Keeping only `tail` suffices, since the head is always `tail.next`: one reference reaches both ends in *O(1)*. A separate `head` field would need updating everywhere — needless work and another source of bugs.

```
public class CircularLinkedList {
    class Node {
        int data;
        Node next;
        Node(int data) {
            this.data = data;
            this.next = null;
        }
    }
    private Node tail;  // head = tail.next
    private int size;
}
```

The class nests its node class and keeps `tail` and `size`. Two special cases: an empty list has `tail == null`, and a lone node points to itself, `tail.next == tail` — odd-looking but consistent, since that node is both first and last.

Common mistake

Traversing a circular list with `while (p != null)`. A ring has no `null`, so the loop never ends. Stop on returning to the start node or reaching `tail`.

5.5.2 Printing a circular list

```
void print() {
    if (tail == null) return;
    Node current = tail.next;
    while (current != tail) {
        System.out.print(current.data + " ");
        current = current.next;
    }
    System.out.print(current.data + " ");
}
```

An empty list prints nothing. Otherwise the traversal starts at the head, `tail.next`, printing and advancing while `current != tail`. The loop stops on the last node before printing it, so the line after the loop prints it.

Step	`current`	`current != tail`	Printed
start	node(1)	—	—
iteration 1	node(1)	true	1
iteration 2	node(8)	true	8
iteration 3	node(10)	true	10
loop ends	node(16)	false	—
last line	node(16)	—	16

Table 5.19: Trace of `print()` on the ring 1 → 8 → 10 → 16 → (1) (`tail` = node 16).

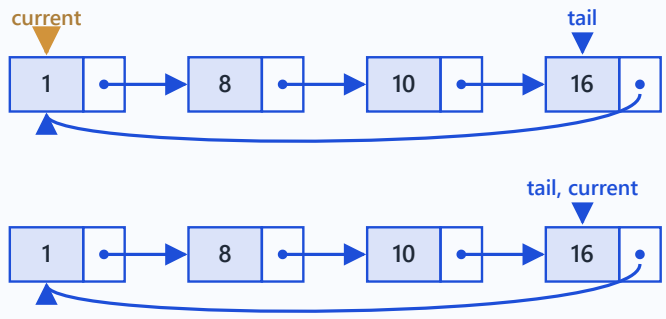


Figure 5.39: Printing a circular list. (a) `current = tail.next`: the traversal starts at the head, 1. (b) When `current` reaches the last node (16), the loop ends; `null` is never reached.

The output is `1 8 10 16`, in *O(n)* time. With one node, `current` equals `tail` at once, the loop is skipped, and the last line prints the node exactly once.

A `do-while` loop also works — print, advance, stop on returning to the head — and since it runs the body before testing, needs no separate last line.

5.5.3 Cycle detection: Floyd's algorithm

A circular list's ring is deliberate, but a bug can create an unwanted cycle in a singly linked list: some `next` points to an earlier node, and printing or measuring the list never ends. So detecting cycles is a practical question.

Watching for a return to the head isn't enough, since the cycle may loop back to a middle node: if 6 points back to 3 in $1 \to 2 \to 3 \to 4 \to 5 \to 6$, the traversal circles 3, 4, 5, 6 forever without revisiting 1. A set of visited nodes works but costs $O(n)$ memory.

Floyd's cycle detection algorithm needs $O(1)$ extra memory. It reuses the slow and fast pointers, hence its nickname, "tortoise and hare."

```
boolean hasCycle(Node head) {
    Node slow = head, fast = head;
    while (fast != null &&
           fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
        if (slow == fast) return true;
    }
    return false;
}
```

Each iteration the slow pointer moves one node and the fast pointer two. Two outcomes are possible:

- **No cycle:** the fast pointer reaches the end — it or its successor becomes `null` — the loop condition fails, and the method returns `false`.
- **A cycle:** both pointers end up circling inside it, and the fast one gains one node per iteration. The gap shrinks by exactly one, so it cannot jump over the slow pointer and must land on it; then `slow == fast` and the method returns `true`.

> **Theorem 5.1 (Correctness of Floyd's algorithm)**
>
> If a singly linked list has a cycle, the slow and fast pointers meet at the same node within n iterations; if not, the fast pointer reaches the end within $n/2$ iterations. Here n is the number of distinct nodes in the list.

Proof. Without a cycle, the fast pointer covers two nodes per iteration and exhausts the path within $n/2$ iterations. With one, let μ be the length of the part before the cycle and λ the cycle length ($\mu + \lambda = n$). The slow pointer enters after μ iterations, with the fast one already inside. Let d be the number of nodes by which the fast pointer then trails the slow one, counting forward around the cycle; $0 \le d < \lambda$.

Each iteration shrinks the gap by exactly one, so they meet after d more, $\mu + d < \mu + \lambda = n$ in total. The exception is $\mu = 0$, a cycle back to the head: then $d = 0$ at the start, but equality isn't tested before the first move, and the pointers coincide again after $\lambda = n$ iterations, still within n. □

Example 5.7

The list is 1 → 2 → 3 → 4 → 5 → 6, and 6 points back to 3.

1. **Iteration 1.** `slow` reaches 2, `fast` 3. 2 ≠ 3.
2. **Iteration 2.** `slow` reaches 3, `fast` 5. 3 ≠ 5.
3. **Iteration 3.** `slow` reaches 4; `fast` goes 5 → 6 and, via the cycle link, to 3. 4 ≠ 3. The fast pointer has lapped the cycle and trails by one node.
4. **Iteration 4.** `slow` reaches 5; `fast` goes from 3 to 4 to 5. 5 == 5: the method returns `true`.

Here $\mu = 2$ (1 and 2) and $\lambda = 4$ (3, 4, 5, 6). The slow pointer enters in iteration 2, with the fast one at 5, two nodes behind it; they meet two iterations later.

Iteration	`slow`	`fast`	`slow == fast`
start	1	1	(not tested)
1	2	3	no
2	3	5	no
3	4	3	no
4	5	5	yes → `true`

Table 5.20: Trace of `hasCycle` (1 → … → 6, 6 → 3).

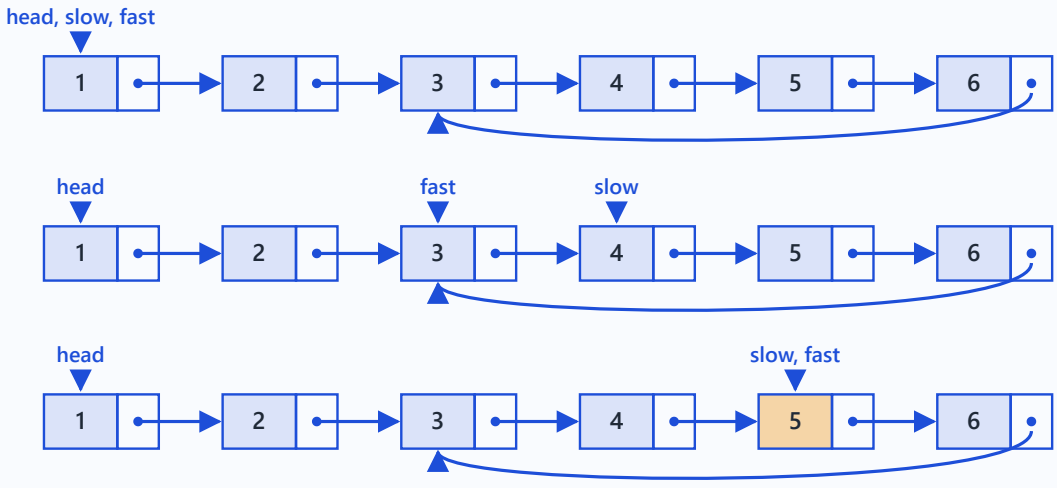


Figure 5.40: Floyd's algorithm. (a) Both pointers start at the head; the link from 6 back to 3 forms the cycle. (b) At the end of iteration 3, the fast pointer has lapped the cycle to 3; the slow one is at 4. (c) In iteration 4

they meet at 5.

At most *n/2* iterations without a cycle and *n* with one: *O(n)* time. Only two pointers: *O(1)* extra memory.

Common mistake

Comparing before the pointers move. Both start at `head`, so the first test succeeds and every list, cyclic or not, gets `true`. Test *after* advancing.

5.5.4 Inserting at the end

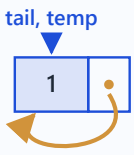
Appending puts the new node between the last node and the head:

```
Node temp = new Node(value);
if (tail == null) {
    tail = temp;
    tail.next = tail;
} else {
    temp.next = tail.next;
    tail.next = temp;
    tail = temp;
}
size++;
```

An empty list's new node becomes the tail and points to itself (`tail.next = tail`), closing the ring. Otherwise: the new node points to the head (`temp.next = tail.next`), the old tail to the new node (`tail.next = temp`), and `tail` moves. `size` grows either way.

Call	Assignments	`tail`	Ring	`size`
`addLast(1)`	`tail = temp`; `tail.next = tail`	node(1)	1 → (1)	1
`addLast(8)`	8 → 1; 1 → 8; `tail` = 8	node(8)	1 → 8 → (1)	2
`addLast(10)`	10 → 1; 8 → 10; `tail` = 10	node(10)	1 → 8 → 10 → (1)	3

Table 5.21: Trace of `addLast(1)`, `addLast(8)`, `addLast(10)` on an empty circular list.



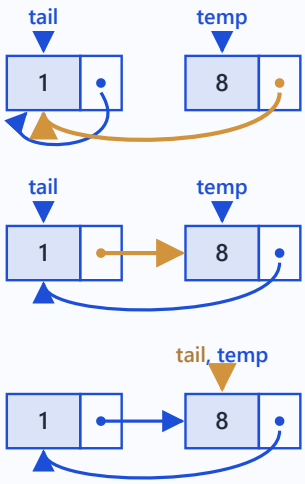


Figure 5.41: Inserting into a circular list. (a) `addLast(1)` : the single node points to itself. (b) `addLast(8)` , first assignment: the new node points to the head (1). (c) Second assignment: the old tail (1) points to the new node. (d) `tail` moves to 8.

At most three assignments, no traversal: *O(1)*. Order matters: doing `tail.next = temp` first erases the only link to the head, so `temp.next = tail.next` links the node to itself and the old nodes fall out of the ring.

5.5.5 Inserting and removing at the front

Front and end insertion are nearly identical here: the new node enters the ring at the same spot, between the last node and the head. Only the choice of tail differs:

```
void addFirst(int value) {
    Node temp = new Node(value);
    if (tail == null) {
        tail = temp;
        tail.next = tail;
    } else {
        temp.next = tail.next;
        tail.next = temp;
    }   // tail is unchanged
    size++;
}
```

This is end insertion minus `tail = temp` : with `tail` unmoved, the new node follows the last node, at the front. `addFirst(5)` on 1 → 8 → 10 → (1) (`tail` = 10) points 5 at the old head, then 10 at 5: 5 → 1 → 8 → 10 → (5), `tail` still 10.

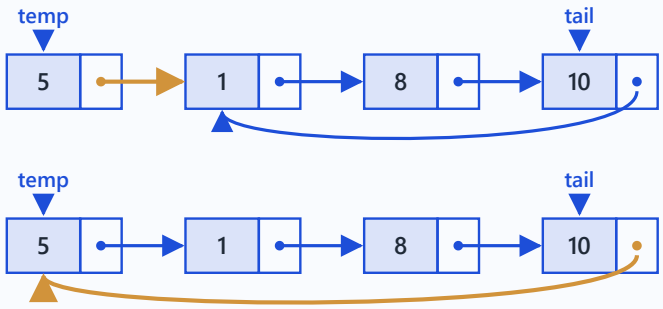


Figure 5.42: `addFirst(5)` on a circular list. (a) `temp.next = tail.next` : 5 points to the old head (1). (b) `tail.next = temp` : 10 now points to 5, the new head; `tail` doesn't change.

Removing from the front is constant time as well:

```
void removeFirst() {
    if (tail == null) return;
    Node head = tail.next;
    if (head == tail)   // single node
        tail = null;
    else
        tail.next = head.next;
    size--;
}
```

The head is `tail.next`. A single node (`head == tail`) empties the list. Otherwise `tail.next = head.next` routes the ring past the head, whose successor becomes the new head: 1 → 8 → 10 → (1) becomes 8 → 10 → (8).

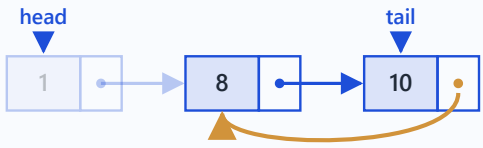


Figure 5.43: Removing from the front of a circular list. `tail.next = head.next` : 10 now points to 8, and 1 has left the ring.

Both are *O(1)*. The local `head` is not a field, just a temporary reference to the front node.

5.5.6 Removing from the end

With only `tail`, removing from the end of a singly linked ring is still *O(n)*: the predecessor's link must change, and reaching it means walking the ring:

```
void removeLast() {
    if (tail == null) return;
    if (tail.next == tail) tail = null;   // single node
    else {
        Node temp = tail.next;
        while (temp.next != tail)     // second to last
            temp = temp.next;
        temp.next = tail.next;     // link to the head
        tail = temp;
    }
    size--;
}
```

A single node (`tail.next == tail`) empties the list. Otherwise `temp` walks from the head until its successor is `tail` ; `temp.next = tail.next` then links it to the head, dropping the old tail, and `tail` moves to `temp` .

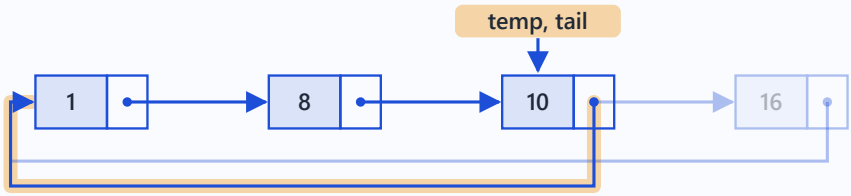
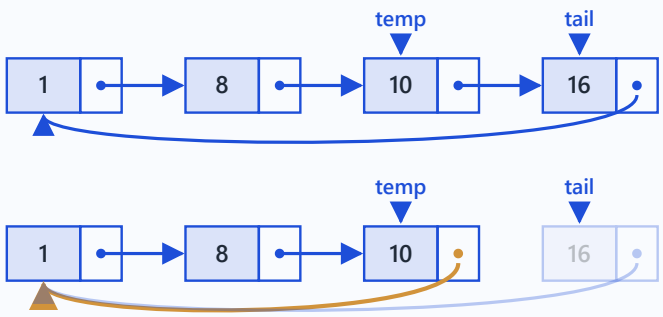


Figure 5.44: Removing from the end of the ring 1 → 8 → 10 → 16. `temp` and the new `tail` are at 10; 10's link is redirected to the head (1) (highlighted link), and 16 leaves the ring (faded).

Step	`temp`	`temp.next == tail`	Decision
start	node(1)	no (8)	advance
advance 1	node(8)	no (10)	advance
advance 2	node(10)	yes (16)	stop
relink	10.next = 1; `tail` = 10; `size` 3: ring 1 → 8 → 10 → (1)		

Table 5.22: Trace of `removeLast()` on the ring 1 → 8 → 10 → 16 → (1) (`tail` = node 16).



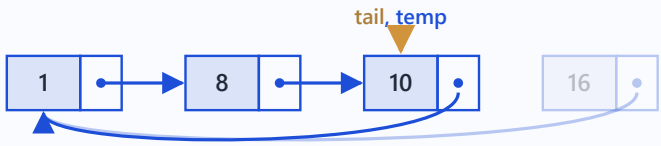


Figure 5.45: The last steps of `removeLast()`. (a) 10's successor is `tail`: `temp` is on the second-to-last node. (b) `temp.next = tail.next`: 10 points to the head, and 16 leaves the ring. (c) `tail = temp`: the tail is now 10.

The loop runs $n - 2$ times: $O(n)$. A doubly linked ring reaches the second-to-last node as `tail.prev` and removes in $O(1)$, which is why many implementations, Java's `LinkedList` among them, link both ways.

5.5.7 Rotation and taking turns

Rotation is the circular list's most elegant operation: moving the front element to the back changes no link; `tail` just advances one node:

```
void rotate() {              // Goodrich: rotate()
    if (tail != null)
        tail = tail.next;    // the old head moves to the back
}
// round-robin: serve the front element each turn
for (int round = 1; round <= 4; round++) {
    System.out.print(tail.next.data + " ");
    list.rotate();
}
```

After `tail = tail.next`, the old head is the tail and its successor, `tail.next`, the new head. The ring is unchanged; only our viewpoint moves. Goodrich and Tamassia call this `rotate()`; it is one assignment, $O(1)$.

Rotation models **round robin** scheduling directly: jobs sharing the processor sit in a ring, the front job gets a short slice, the list rotates, and the next job's turn comes. The code's second part runs four turns.

Turn	Served (`tail.next`)	`tail` after `rotate()`	New head
1	1	1	8
2	8	8	10
3	10	10	1
4	1	1	8

Table 5.23: Four turns of round robin on the ring 1 → 8 → 10 → (1) (initially `tail` = 10).

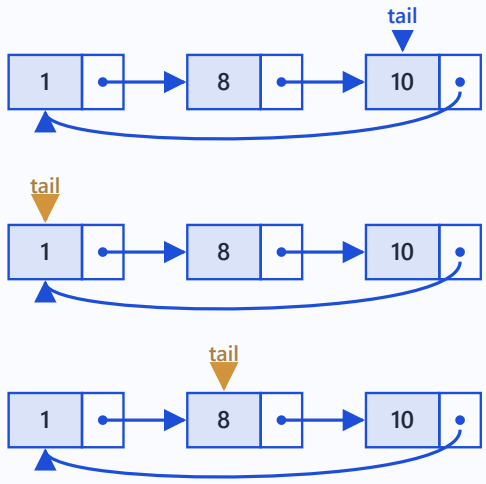


Figure 5.46: Rotation. (a) Start: `tail` = 10, head 1. (b) First `rotate()`: `tail` moves to 1, and the head becomes 8. (c) Second `rotate()`: `tail` moves to 8, and the head becomes 10. No link changed.

The output is `1 8 10 1`: fair turns. A plain list could remove from the front and append instead, building new links and updating the tail; the ring needs one assignment. Finished jobs leave with `removeFirst()` and new ones join with `addLast`, both *O(1)*.

Common mistake

Rotating by detaching and re-appending nodes. It works, but changes links needlessly and risks a wrong `tail`. A ring's starting point is just where `tail` points; advancing it is all rotation takes.

5.6 Array or linked list?

Arrays and linked lists answer the same question — how to store a sequence — and the better choice depends on what the program does. Figure 5.47 shows both in memory.



Figure 5.47: Left, an array: four elements in consecutive cells at addresses 1000, 1004, 1008, and 1012. Right, a linked list: the same four elements in scattered nodes, ordered by arrows.

An array finds element *i* at its start address plus *i* element sizes: with 4-byte integers, index 2 is at *1000 + 2 · 4 = 1008*, in one step regardless of length. A linked list must follow links.

Criterion	Array	Linked list
Access to element *i*	O(1): address computed	O(n): walk from head
Insert / remove at front	O(n): shifting	O(1)
Insert after a known position	O(n)	O(1): two links
Size	fixed; growing is an O(n) copy	dynamic, one `new` per insertion
Overhead and cache	unused capacity; cache-friendly	a reference per node; scattered access

Table 5.24: Arrays compared with singly linked lists.

The last row of Table 5.24 shows what big-O misses. The processor reads memory in consecutive blocks and keeps them in a fast store, the **cache**. Reading one block of an array brings the next few elements along; scattered list nodes may each need a separate read. So a traversal that is *O(n)* for both is often markedly faster on an array.

Both waste memory. An array may leave capacity unused — up to the element count in a doubling dynamic array. A list leaves none but spends a reference and object header per element, more than the data for small elements like integers.

A rough rule of thumb:

- **Array**: when random access is frequent, the element count is roughly known, and elements are mostly appended.
- **Linked list**: when changes cluster at the ends or at known positions, the size is unpredictable, and indexing is rare. Stacks and queues, which work only at the ends, are natural fits.

Which kind of linked list to use also depends on the operations; Table 5.25 compares four common forms.

Operation	Singly (`head` only)	Singly (`head` + `tail`)	Doubly (`head` + `tail`)	Circular (`tail` only)
Insert at front	O(1)	O(1)	O(1)	O(1)
Insert at end	O(n)	O(1)	O(1)	O(1)
Remove from front	O(1)	O(1)	O(1)	O(1)
Remove from end	O(n)	O(n)	O(1)	O(n)
Search / element *i*	O(n)	O(n)	O(n)	O(n)
Remove a known node	O(n)*	O(n)*	O(1)	O(n)*

Table 5.25: Operation costs for the kinds of linked list.

Starred costs come from walking to the predecessor. Inserting *after* a known node is *O(1)* in all four, since only that node's link changes. So if you often remove nodes at the end or in the middle, choose a doubly linked list.

5.7 Common mistakes

Most linked list bugs fall into a few types. The per-operation mistakes are collected here as a review checklist.

Bug	What happens	Fix
No empty-list check	`head.data` or `head.next` while `head` is `null`: `NullPointerException`.	`if (head == null) return;`
Setting links in the wrong order	If `head` or the predecessor's link changes first, the rest of the list is lost; a node may point to itself.	Attach the new node to the rest first: `newNode.next = head; head = newNode;`
Advancing `head` while traversing	The list is unreachable after printing.	Traverse with `Node temp = head;`.
Forgetting `tail`	After removing the only node, `tail` still points to it; after inserting into an empty list, `tail` stays `null`.	`if (head == null) tail = null;`; on insertion into an empty list, `head = tail = newNode;`
Updating one direction in a doubly linked list	If `prev` isn't updated, backward traversal breaks; forward printing hides the bug.	Four links per insertion, two per removal: `next.prev = newNode;`
`!= null` in a circular list	A ring has no `null`: infinite loop.	`while (p != tail) …`
Not updating `size` in some methods	`size()` and `isEmpty()` answer wrongly; the counter may go negative.	Every method that changes the list updates the counter.

Table 5.26: Common bugs in linked list code.

Two questions lie behind all of these. First: **"Which node is unreachable after this assignment?"** Overwriting a link erases a path to its old target; if the target is still needed, save it in a local first, as `next` does in reversal and `temp` in front removal. Second: **"Is this right for an empty list, a one-node list, and the first and last nodes?"** Most bugs surface in these four cases, and tracing a method through them on paper catches most of them. Sentinels are valuable precisely because they remove these cases structurally.

5.8 Summary

- A **linked list** keeps elements in **nodes** connected by references; links, not memory positions, define the order. The node class has a field of its own type (self-referential structure).
- In a **singly linked list** each node points to the next, the last to `null`, and the list is reached through `head`. Front insertion and removal are *O(1)*; search, indexing, appending (without a tail pointer), and removal from the end are *O(n)*.
- The **traversal pattern**: `temp = head`; while `temp != null`, work; `temp = temp.next`. Printing, length, and search are variations; `temp.next != null` stops at the last node.

- The **order of links** is critical: attach the new node to the rest first, then change the predecessor's link or `head`. Walk to the node *before* the one you operate on.
- A **tail pointer** makes appending *O(1)* but not removal from the end; every operation must maintain `tail` (and `size`).
- **Reversal** uses three pointers and no new nodes: *O(n)* time, *O(1)* extra memory. **Slow and fast pointers** find the middle in one pass, returning the second middle for even lengths.
- A **deep copy** creates new nodes; `newList.head = head` is only a second reference. Removing duplicates is $O(n^2)$ with a seen-list, *O(n)* with a hash set. Merging sorted lists is *O(n + m)* with a tail pointer. Value-swap sorting makes $n(n-1)/2$ comparisons; *O(n log n)* merge sort suits lists better.
- In a **doubly linked list** nodes point both ways. Insertion and removal at either end and removal of a known node are *O(1)*, and backward traversal works, at the cost of a second reference per node and two-way updates. Middle insertion changes four links in a fixed order.
- **Sentinel nodes** (header and trailer) give every real node a predecessor and a successor, eliminating `null` checks and `head` / `tail` updates.
- A **linked structure from a matrix** gives each node `right` and `down` links. Built from the bottom right with the nodes stored in a table, it takes *O(rows · columns)*.
- In a **circularly linked list** the last node points to the head; only `tail` is kept, and the head is `tail.next`. Appending, front insertion and removal, and **rotation** are *O(1)*; removal from the end is *O(n)*. Traversal stops at `tail`, not `null`. Rotation models round robin scheduling.
- **Floyd's algorithm** uses slow and fast pointers to detect a cycle in *O(n)* time and *O(1)* extra memory.
- **Arrays** win at random access and cache use; **linked lists** win at insertion and removal at the ends and at unpredictable sizes.

5.9 Key terms

Each term links to the course glossary, which gives its Turkish equivalent and its use in other chapters.

linked list · node · head · tail · pointer · reference · null · singly linked list · doubly linked list · circularly linked list · traversal · sentinel · cycle detection · abstract data type (ADT) · generic · deep copy · shallow copy · swap · merge sort · hash table · matrix · round robin · time complexity · space complexity

5.10 Test Yourself

1. On an empty `SinglyLinkedList` (keeping `head` and `tail`) we call `addLast(4)`, `addFirst(2)`, `addLast(9)`, `addAt(7, 1)`, and `removeFirst()`. Which value does the node returned by the last call hold, and what is the final list?

   Answer

   The list goes 4; 2 → 4; 2 → 4 → 9; 2 → 7 → 4 → 9. In `addAt(7, 1)` the loop runs *index − 1 = 0* times, so 7 goes in after the leading 2. `removeFirst()` returns 2's node, leaving 7 → 4 → 9 with `tail` still 9.

2. What do `search(6)` and `search(5)` return on 6 → 3 → 6 → 1? How many nodes' data does each compare?

   Answer

   `search(6)` matches the first node and returns 0 after one comparison, never reaching the second 6. `search(5)` checks all four nodes and returns −1. These are the best (*O(1)*) and worst (*O(n)*) cases.

3. `reverse()` runs on 5 → 2 → 7. At the end of the second iteration, what are `prev`, `current`, and the reversed part? What are the final list and `tail`?

   Answer

   Iteration 1 turns 5's link to `null` (`prev` 5, `current` 2); iteration 2 turns 2's link to 5 (`prev` 2, `current` 7). The reversed part is 2 → 5 → null; 7's link is still `null`. After iteration 3, `head = prev` gives 7 → 2 → 5. The first line set `tail` to the old head, 5.

4. What does `findMiddle()` return for 3 → 1 → 4 → 1 → 5 and for 9 → 8 → 7 → 6? What must change to get the first middle of the second list?

   Answer

   First list, (slow, fast) per iteration: (1, 4), (4, 5); `fast.next` is then `null`, and 4 is returned. Second list: (8, 7), (7, null), returning 7, the second middle. For the first middle (8), use `fast.next != null && fast.next.next != null`; one iteration runs and `slow` stays at 8.

5. `removeDuplicates()` runs on 2 → 2 → 5 → 2 → 5 → 9. What is the result, how many nodes are

skipped, how many times is `contains` called, and what does `seen` hold at the end?

Answer

The result is 2 → 5 → 9. The second, fourth, and fifth nodes (2, 2, 5) are skipped: 3 nodes. `contains` runs once per node, 6 times. `seen` grows at the front, so it holds 9 → 5 → 2. When the second node goes, `prev` stays on the first, so the fourth node's 2 is also skipped correctly after 5.

6. 1 → 5 → 6 and 2 → 3 → 7 → 8 are combined with `merge`. How many comparisons does the main loop make, and which elements are appended without comparison?

Answer

Comparisons: 1 ≤ 2 (1 appended), 5 ≤ 2 false (2), 5 ≤ 3 false (3), 5 ≤ 7 (5), 6 ≤ 7 (6). The first list is exhausted: 5 comparisons. 7 and 8 are appended without comparison, giving 1 → 2 → 3 → 5 → 6 → 7 → 8.

7. On the doubly linked list 3 ⇄ 6 ⇄ 9 we call `remove(3)`, `removeLast()`, and `addFirst(1)`. Which branch of `remove` and of `removeLast` runs in each call? What do forward and backward reading give at the end?

Answer

`remove(3)`: 3 is the head (`temp.prev` is `null`), so `else head = temp.next` runs; the right neighbor 6 exists, so 6.prev becomes null: 6 ⇄ 9. `removeLast()`: `head != tail`, general branch; `tail` moves to 6, 6.next = null. `addFirst(1)`: 1 ⇄ 6. Forward: 1 6; backward: 6 1.

8. In 1 → 2 → 3 → 4 → 5 → 6 → 7, node 7 points back to 5. In which iteration, and at which node, does `hasCycle` return `true`? On the cycle-free list 1 → 2 → 3 → 4 → 5, after how many iterations does it return `false`?

Answer

(slow, fast) per iteration: (2, 3), (3, 5), (4, 7), (5, 6), (6, 5), (7, 7) — they meet at 7 in iteration 6. Here $\mu = 4$, $\lambda = 3$: when the slow pointer enters in iteration 4, the fast one is at 6, two behind, and they meet two iterations later. Without a cycle the fast pointer goes 1 → 3 → 5; after iteration 2, `fast.next` is `null` and the method returns `false`.

9. On the circular list 4 → 7 → 2 → (4) (`tail` = 2) we call `removeLast()` and then `addFirst(9)`. How many times does the loop in `removeLast` advance? What does `print()` output at the end,

and which node is `tail`?

Answer

`temp` starts at 4; its successor 7 isn't `tail`, so it advances once, and 7's successor is `tail`. 7 is linked to 4 and becomes `tail`: 4 → 7 → (4). `addFirst(9)`: 9 points to 4, 7 to 9, `tail` unchanged. `print()` outputs `9 4 7`; `tail` is 7.

10. A 3×3 matrix is turned into a linked structure with `build`. How many nodes are created, and how many `right` and `down` links are non-null? How many nodes does a recursive version create that calls `build(i, j+1)` and `build(i+1, j)` directly, without the table `d`?

Answer

`build` creates 9 nodes, with $3 \cdot 2 = 6$ `right` and $2 \cdot 3 = 6$ `down` links. The recursive version creates a cell once per right-down path from the top left: row by row 1, 1, 1 / 1, 2, 3 / 1, 3, 6, or 19 nodes.

5.11 Further reading

- M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, *Data Structures and Algorithms in Java* (6th ed., 2014), Chapter 3: singly linked, circular, and doubly linked lists with sentinels, in Java code very close to the ADT in these notes; `rotate()` and list copying are covered there too.
- R. Sedgewick and K. Wayne, *Algorithms* (4th ed., 2011), Section 1.3: linked lists as the basis for stack, queue, and bag implementations, with short, tidy code.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms* (3rd ed., 2009), Section 10.2: doubly linked lists and circular lists with a single sentinel, in pseudocode and more formal language.
- D. E. Knuth, *The Art of Computer Programming*, Vol. 2 (3rd ed., 1997), Section 3.1, Exercise 6: among the first written sources to credit Floyd with cycle detection by slow and fast pointers.

5.12 References

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 3rd ed., MIT Press, 2009.
2. M. T. Goodrich, R. Tamassia, M. H. Goldwasser, *Data Structures and Algorithms in Java*, 6th ed., Wiley, 2014.
3. D. E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, 3rd ed., Addison-Wesley, 1997.
4. R. Lafore, *Data Structures and Algorithms in Java*, 2nd ed., Sams, 2002.
5. R. Sedgewick, K. Wayne, *Algorithms*, 4th ed., Addison-Wesley, 2011.

PART II · CHAPTER 6

The List ADT and Java Collections

The list is the data structure programmers use most. Student names, the items in a cart, the moves of a game: all are kept in order and reached by position. Earlier chapters treated arrays and linked lists separately; this one studies the abstract list that sits above both, and how Java organizes it. We start with the interfaces `Iterable`, `Collection`, and `List`, then the list ADT's operations, views, and unmodifiable lists, the `List` methods, and three traversal tools: `Iterator`, `ListIterator`, and `Spliterator`. Later sections open up `ArrayList` and `LinkedList`, work out an amortized cost, and introduce the positional list; `Vector`, `Stack`, `CopyOnWriteArrayList`, and `Arrays.asList` close the chapter.

Code for this chapter: `github.com/sercankulcu/data-structures-java/Bolum06/src`

LEARNING OUTCOMES

The list ADT
Explain the basic operations of an ordered collection (append, insert, remove, index, slice, search, length) and name the Java `List` method for each.

Traversal
Distinguish the roles of `Iterator`, `ListIterator`, and `Spliterator`; remove safely while traversing and explain what causes a `ConcurrentModificationException`.

Implementations
Compare the internals and operation costs of `ArrayList` and `LinkedList`; compute the amortized cost of a growing array.

Choosing an API
Pick the right list class and method for a task, minding the difference between a view, a copy, and an unmodifiable list.

6.1 Collection interfaces

A data structure can be described at two levels: *what* it does (which operations exist, what each changes and returns) and *how* (how the data sits in memory, which steps each operation takes). The first is an **abstract data type** (ADT); the concrete structure at the second level is an **implementation** of it.

Java builds this split into the language. An ADT's operations go in an **interface**, which gives method names, parameters, and return types but no bodies; the code that does the work lives in a **class** that implements the interface. Java's ready-made data structures follow this pattern in the **Java Collections Framework**, package `java.util`. This section walks down its list side.

6.1.1 The Java Collections Framework: the list side

A **collection** is an object holding a group of elements; lists, sets, and queues are collections. The framework's interfaces form an inheritance tree: the top asks for the least, and each step down adds abilities. Figure 6.1 shows its list side.

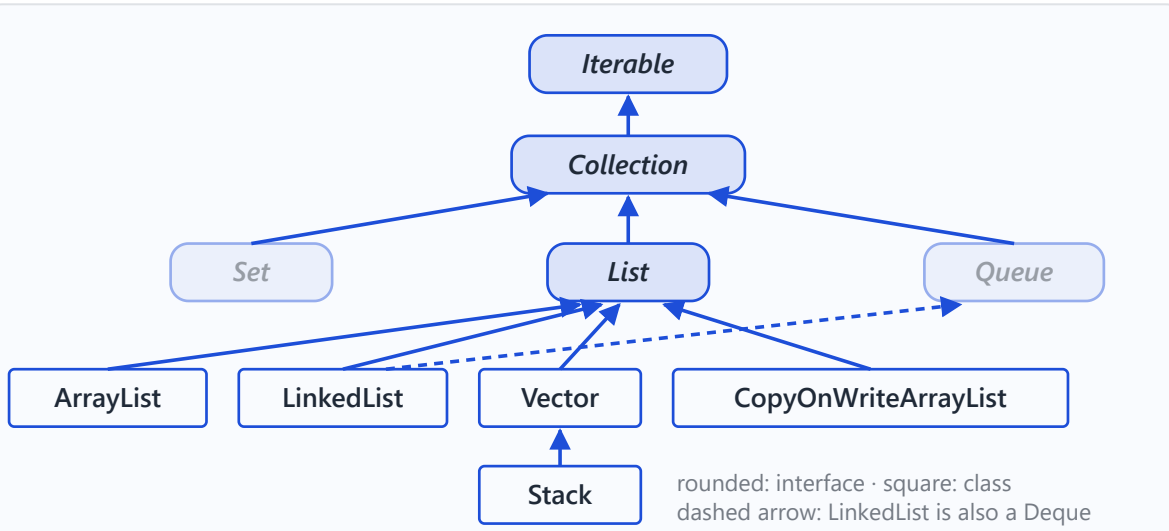


Figure 6.1: The list side of the Java collection hierarchy. Rounded boxes are interfaces, square boxes classes. `Collection` extends `Iterable`; `List`, `Set`, and `Queue` extend `Collection`. `ArrayList`, `LinkedList`, `Vector`, and `CopyOnWriteArrayList` implement `List`; `Stack` extends `Vector`. The dashed arrow: `LinkedList` is also a `Deque`, hence a `Queue`.

Two verbs describe the arrows. An interface *extends* another: `List` inherits all of `Collection`'s methods and adds more. A class *implements* an interface: `ArrayList` supplies a body for every method of `List`. So an `ArrayList` is at once a `List`, a `Collection`, and an `Iterable`.

Interface versus class is ADT versus data structure: *what* versus *how*. `List` says "get the element at an index"; `ArrayList` reads an array cell, `LinkedList` walks nodes. Code that knows only the interface works whichever implementation runs.

Hence the Java habit of declaring the variable with the interface type and creating the object from a concrete class:

```
List<String> l = new ArrayList<>();
```

The left side says "`l` is a list," the right side "for now, an `ArrayList`." If the program turns out to insert at the front a lot, only the right side becomes `new LinkedList<>()`; the hundreds of lines using `l` stay. Had `l` been declared `ArrayList<String>`, some line might use an `ArrayList`-only method such as `trimToSize()`, and the change would not compile.

Common mistake

Creating an object from an interface: `List<String> c = new List<>();` does not compile, because an interface has no method bodies to run. `new` must be followed by a class that implements the interface. `Iterable<String> d = new ArrayList<>();` does compile, since `ArrayList` also implements `Iterable`.

6.1.2 Iterable and the for-each loop

At the top, `Iterable` asks only that an object's elements can be visited in order. It offers three things:

- `iterator()` returns an `Iterator` over the elements. It is the only abstract method; writing it makes a class `Iterable`.
- `spliterator()` returns a `Spliterator`, which splits the elements into parts for parallel processing.
- `forEach(...)` runs a given action (a lambda expression) on each element.

The last two have default bodies built on `iterator()` or a similar tool, so the real contract of `Iterable` is: "Ask me, and I will hand you an iterator."

An **iterator** is a separate object that hands out a collection's elements one at a time and remembers where it is. It answers "Is there another?" (`hasNext()`) and "Give me the next" (`next()`). The for-each loop is built on these two; we write the short form, and the compiler produces the long one:

```
// what we write
for (var x : a) {
    System.out.println(x);
}
```

```
// what the compiler produces (if a is Iterable)
Iterator<T> it = a.iterator();
while (it.hasNext()) {
    var x = it.next();
    System.out.println(x);
}
```

The compiler checks the type of `a`. If `a` is `Iterable`, it produces the second fragment: call `a.iterator()`, then, while `hasNext()` is true, fetch an element with `next()` and run the body. If `a` is an array, it produces an ordinary indexed loop from `i = 0` to `a.length - 1`.

So a for-each loop compiles only over an array or an object of a class that implements `Iterable`; to loop over our own class with for-each, the class must implement `Iterable`.

6.1.3 Code trace: an Iterable class of our own

Here is a collection that stores no elements at all: `Range(2, 5)` stands for 2, 3, and 4, the half-open interval *[2, 5)*, and the iterator generates the numbers as it goes.

```
class Range implements Iterable<Integer> {
  private final int start, end;
  Range(int s, int e) { start = s; end = e; }
  public Iterator<Integer> iterator() {
    return new Iterator<Integer>() {
      int k = start;
      public boolean hasNext() { return k < end; }
      public Integer next() {
        if (k >= end)
          throw new NoSuchElementException();
        return k++;
      }
    };
  }
}
```

Line by line:

- `class Range implements Iterable<Integer>`: an `Iterable` of `Integer`s. The type argument is `Integer` because a type parameter cannot be primitive.
- The bounds `start` and `end` are `final`; they never change.
- Each `iterator()` call returns a *new* iterator, an object of the **anonymous inner class** `new Iterator<Integer>() { … }`. Its field `k` starts at `start`, and it can read the enclosing `Range`'s fields.
- `hasNext()` checks that the next value is in range: `k < end`.

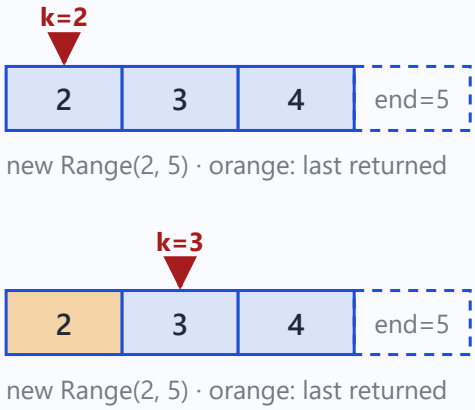
- `next()` first guards the contract: with nothing left (`k >= end`) it throws `NoSuchElementException`. Otherwise `return k++` returns the current `k` *and then* increments it.

Now trace `for (int x : new Range(2, 5))`. Since `x` is an `int`, each `Integer` from `next()` is unboxed.

Step	Call	Check	Returns	`k` (after)	Body
1	`new Range(2, 5)`	—	object	—	—
2	`iterator()`	—	iterator	2	—
3	`hasNext()`	2 < 5	`true`	2	—
4	`next()`	2 >= 5? no	2	3	prints `x = 2`
5	`hasNext()`	3 < 5	`true`	3	—
6	`next()`	3 >= 5? no	3	4	prints `x = 3`
7	`hasNext()`	4 < 5	`true`	4	—
8	`next()`	4 >= 5? no	4	5	prints `x = 4`
9	`hasNext()`	5 < 5	`false`	5	loop ends

Table 6.1: Trace of `for (int x : new Range(2, 5))`. `k` is the iterator's field.

The output is 2, 3, 4; the range is open on the right, so 5 never appears. The iterator's whole state is `k`, and the `Range` never changes. Two for-each loops over one `Range` each call `iterator()` and start from 2 with their own `k`.



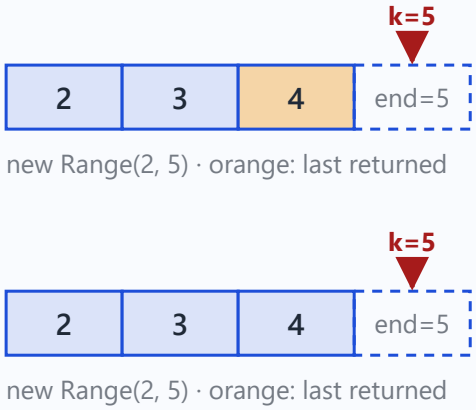


Figure 6.2: The iterator over `Range(2, 5)` at four moments. (a) After `iterator()`, `k = 2`. (b) The first `next()` returned 2 (orange); `k = 3`. (c) The third `next()` returned 4; `k = 5`. (d) `hasNext()` finds `5 < 5` false and the loop ends.

Calling `next()` once more after the loop finds `k = 5` and `end = 5`; the check `k >= end` holds and `NoSuchElementException` is thrown, as the `Iterator` contract requires when no elements remain.

> **Common mistake**
>
> Dropping the `k >= end` check in `next()` because "`hasNext()` is called first anyway." A for-each loop does call it first, but code using an iterator by hand need not. Without the check, `next()` would silently return 5, 6, 7. An iterator that gives wrong answers quietly is far harder to catch than one that throws.

6.1.4 The Collection interface

Below `Iterable` sits `Collection`. Lists, sets, and queues behave very differently but share a core, gathered in this interface: add, remove, count, and test whether an element is present.

Ability	Methods	Example
Adding / removing elements	`add(e)`, `remove(o)`, `clear()`	`c.add("Ann")` → `true`
Size and emptiness	`size()`, `isEmpty()`	`c.size()` → 1
Membership test	`contains(o)`	`c.contains("Ann")` → `true`
Stream processing	`stream()`	`c.stream().count()`
Copying to an array	`toArray()`	`c.toArray(new String[0])`
Bulk operations	`addAll`, `removeAll`, `removeIf`	`c.removeIf(s -> s.isEmpty())`

Table 6.2: What the `Collection` interface offers. In the Example column, `c` is an initially empty `Collection<String>`.

Mind the return values. `add(e)` returns `true` if the collection changed: always for a list, which accepts duplicates, while a set that already holds the element returns `false`. `remove(o)` returns `true` if it found and removed the element.

`stream()` turns the elements into a **stream**, on which filtering, mapping, and counting are chained. `toArray(new String[0])` copies the elements into an array; the empty argument only fixes the result's type. `removeIf` removes every element satisfying a condition (a lambda) in one pass.

What `Collection` lacks matters too: there is no **index**. A set may have no order, so "the third element" is meaningless, and position-based operations such as `get(i)` live one level down, in `List`.

Common mistake

Forgetting the variable's type. After `Collection<String> c = new ArrayList<>(List.of("x", "y"));`, `c.get(0)` does not compile: the object is an `ArrayList` with a `get`, but the compiler looks only at the variable's type, `Collection`. An interface type buys flexibility at the price of using only that interface's methods.

6.1.5 The List interface

`List`, in `java.util`, extends `Collection` with order. Its documentation calls it an *ordered collection* or *sequence*; "ordered" means the elements have a *positional order*, not that they are sorted.

Definition 6.1

A **list** is an abstract data type that keeps elements in positional order and accesses each through an integer **index** starting at 0. The user decides where an element goes; elements stay put and shift only when something is inserted or removed before them. Duplicates are allowed.

Its main properties:

- **Order is preserved.** Elements added as "Ann", "Dan", "Ben" come back in that order.
- **The user picks the position.** `add(e)` appends; `add(i, e)` inserts at a position.
- **Indexed access.** `get(i)`, `set(i, e)`, and `remove(i)` work by position, from 0.
- **Duplicates are allowed.** Most implementations also accept `null`.
- **Search.** `indexOf(o)` gives the first occurrence; `contains(o)` tests presence.

Think of a to-do list. Items are written one below another and named by position; a new item can be squeezed in, a finished one crossed out, and the same item ("drink water") written twice. `List` mimics exactly this.



Figure 6.3: A to-do list: items in positional order; items can be inserted between others and repeated.

`List` is an interface too, so `new List()` is impossible; choose a class. Java's main ones are `ArrayList`, `LinkedList`, `Vector`, `Stack`, and `CopyOnWriteArrayList`. They share the methods, but different internals make the same operation cost different amounts.

6.2 The list ADT

First the list in the abstract: the concepts that describe it and the operations it offers. None of this depends on a language; Python's `list`, C#'s `List<T>`, and Java's `List` are faces of one ADT.

6.2.1 Basic concepts

Four concepts recur; Figure 6.4 shows them in one picture.

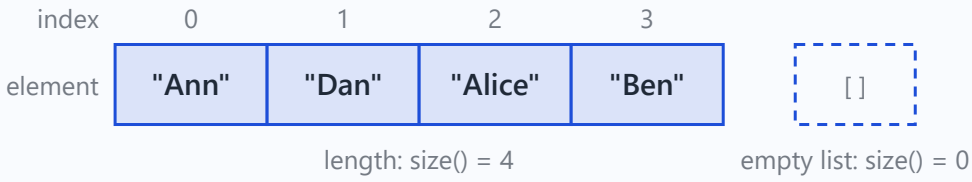


Figure 6.4: A four-element list and an empty list. The top-row numbers 0–3 are indices, the names in the boxes elements. The left list has length 4, the empty one length 0.

Element

A piece of data in the list: a name, a number, an object. Java list elements are always objects; primitives are boxed into wrapper classes.

Index

The integer giving an element's position: 0 for the first, *length − 1* for the last. In Figure 6.4, "Alice" has index 2.

Empty list

A list with no elements; `isEmpty()` returns `true`. It is not `null`: the empty list exists and its methods can be called, while `null` refers to no object.

Length

The number of elements, `size()` in Java. Not to be confused with **capacity**, the number of cells reserved in an array-based list's internal array, which is usually larger.

On a bus, the seats are the capacity and the seated passengers the length. Users see only the passengers; empty seats are the list's internal business.

6.2.2 List operations and their Java methods

The list ADT has seven basic operations; Table 6.3 pairs each with its Java `List` method.

ADT operation	Meaning	Java `List` method
Append	adds a new element at the end	`add(e)`
Insert	adds at a given position; later elements shift	`add(i, e)`
Remove	takes out the element at a position, or a given element	`remove(i)`, `remove(o)`
Indexing	reads or replaces the element at a position	`get(i)`, `set(i, e)`
Slicing	selects a range: *[start, end)*	`subList(from, to)`
Search	finds an element's position	`indexOf(o)`; −1 if absent
Length	returns the number of elements	`size()`

Table 6.3: List ADT operations and Java `List` methods.

"Later elements shift" is the heart of a list. Inserting `X` at position 1 of `[A, B, C]` gives `[A, X, B, C]`: B moves from index 1 to 2, C from 2 to 3. Removal does the reverse. A list has no gaps; its indices run unbroken from 0 to *length − 1*.

[start, end) is Java's **half-open interval** convention: start included, end excluded. `subList(1, 3)` selects indices 1 and 2, *3 − 1 = 2* elements. A slice's length is one subtraction, and `subList(0, size())` is the whole list.

`subList` differs from the other operations in one way: it returns not a **copy** but a **view** of the original.

> **Definition 6.2**
>
> A **view** stores no elements of its own and operates on those of another collection. Changes through the view show up in the underlying collection, and changes to that collection are visible through the view.

Hence a short idiom: `l.subList(1, 3).clear()` removes indices 1 and 2 from the original, with no "remove range" method needed.

6.2.3 Code trace: subList is a view

A view on a five-element `ArrayList`, modified:

```
List<String> l = new ArrayList<>(
    List.of("A", "B", "C", "D", "E"));
List<String> sub = l.subList(1, 3);
System.out.println(sub);
sub.set(0, "X");
sub.add("Y");
sub.clear();
System.out.println(l);
```

`subList(1, 3)` allocates no array and copies nothing. The view holds a reference to the original, an offset, and a length — here 1 and *3 − 1 = 2*. `sub.get(j)` reads `l.get(1 + j)`, and every operation is translated the same way.

Line	Original list `l`	View `sub`	View length	Note
`new ArrayList<>(…)`	[A, B, C, D, E]	—	—	5 elements
`l.subList(1, 3)`	[A, B, C, D, E]	[B, C]	2	no copy; *[1, 3)*
`println(sub)`	[A, B, C, D, E]	[B, C]	2	reads `l[1]`, `l[2]`; output `[B, C]`
`sub.set(0, "X")`	[A, X, C, D, E]	[X, C]	2	`l.set(1 + 0, "X")`
`sub.add("Y")`	[A, X, C, Y, D, E]	[X, C, Y]	3	`l.add(1 + 2, "Y")`; D, E shift right
`sub.clear()`	[A, D, E]	[ ]	0	indices 1–3 of `l` removed
`println(l)`	[A, D, E]	[ ]	0	output `[A, D, E]`

Table 6.4: Trace of `subList`. The view's `offset` is 1 throughout.

Three moments deserve a closer look.

Reading and writing. `sub.set(0, "X")` becomes `l.set(1, "X")` once the offset is added, so X replaces B in the original. The view keeps no copy, so printing it now gives `[X, C]`.

Adding. `sub.add("Y")` appends at the view's end, position *offset + length = 1 + 2 = 3* of the original: `l.add(3, "Y")`. D and E shift right, and the view grows to length 3.

Clearing. `sub.clear()` removes X, C, and Y (indices 1–3) from the original; D and E shift three places left. The original becomes `[A, D, E]`, the view empty.

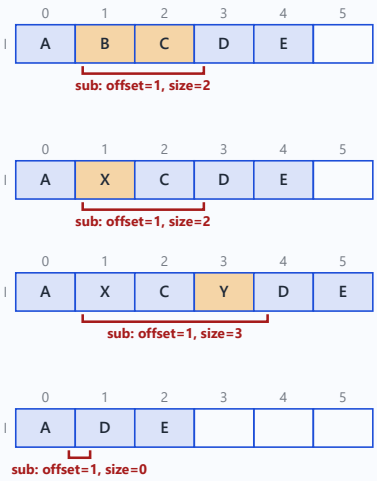


Figure 6.5: The `subList` trace at four moments; the bracket marks the view's range. (a) `subList(1, 3)`: offset 1, length 2. (b) `sub.set(0, "X")` writes cell 1 of the original. (c) `sub.add("Y")` inserts at index 3; the view has 3 elements. (d) After `sub.clear()`: original `[A, D, E]`, view empty.

A view's power brings a danger. Its arithmetic is relative to the original as it was when the view was opened. If the original is changed *structurally* (elements added or removed) other than through the view, the offset and length may be wrong, so Java makes the stale view throw `ConcurrentModificationException` on its next use. The check uses the same `modCount` counter as iterators (Section 6.4.4).

Example 6.1

Let `l = [A, B, C, D]`. After `List<String> v = l.subList(1, 3); v.clear();` and `l.add("E")`, what does `v.size()` do?

1. **The view.** `v` has offset 1 and length 2: `[B, C]`.
2. **Clearing.** `v.clear()` removes indices 1 and 2: `l = [A, D]`, `v` empty. The removal went through the view, which updated its bookkeeping and stays valid.
3. **Adding from outside.** `l.add("E")` bypasses the view: `l = [A, D, E]`.
4. **Using the stale view.** `v.size()` notices the original changed behind its back and throws `ConcurrentModificationException`.

Common mistake

Assuming `subList` returns an independent list. Code that writes `List<String> firstThree = l.subList(0, 3);` and then modifies `firstThree` modifies `l` too. For an independent piece, copy the view: `new ArrayList<>(l.subList(0, 3))`.

6.2.4 Modifiable and unmodifiable lists

So far every list could be changed freely after creation. Not all can; Java has two kinds.

Mutable list

Elements can be added, removed, and replaced after creation: `new ArrayList<>()`, `new LinkedList<>()`.

Immutable list

Contents never change after creation; a different list means a new list: `List.of(...)`, `List.copyOf(...)`.

The two side by side:

```
List<String> fixed = List.of("Ann", "Dan");
fixed.add("Ben");    // UnsupportedOperationException
List<String> copy = new ArrayList<>(fixed);   // mutable copy
copy.add("Ben");  // [Ann, Dan, Ben] — fixed is still [Ann, Dan]
```

`List.of("Ann", "Dan")` is immutable. The variable's type is still `List<String>`, so `fixed.add("Ben")` *compiles*; the error surfaces only at run time as `UnsupportedOperationException`. `fixed.set(0, "X")` throws the same. `List.of` also rejects `null`: `List.of("a", null)` throws `NullPointerException`.

`new ArrayList<>(fixed)` gives a modifiable version by copying the elements into a new array. The two objects are separate: at the end `copy` is `[Ann, Dan, Ben]` and `fixed` still `[Ann, Dan]`.

Why want a structure that cannot change? When many parts of a program share a list, the restriction becomes a guarantee: no method that receives it can corrupt anyone else's data, and threads can read it without locks, since it cannot change mid-read. Immutability is a basic tool of functional programming.

A more advanced form is the **persistent data structure**, where every "update" makes a new version and old versions stay usable. Rather than copying everything, it shares the unchanged parts and rebuilds only the changed path. Adding to the front of a linked list, for example, creates one new node pointing to the untouched old list; the two share the tail.

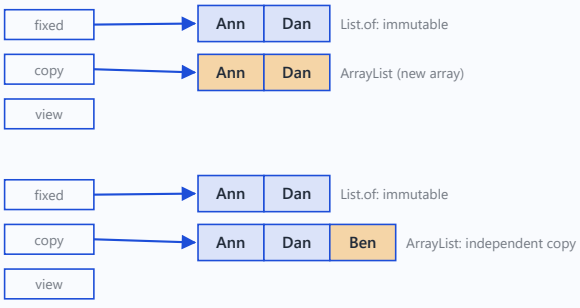
A third option is often confused with these: `Collections.unmodifiableList(l)`. It does *not* make an immutable list; it returns a **read-only view** wrapped around `l`. Writing through it is forbidden, but when `l` changes, so does the view.

6.2.5 Code trace: copy, view, or unmodifiable?

```
List<String> fixed = List.of("Ann", "Dan");
// fixed.add("Ben");     → error
List<String> copy = new ArrayList<>(fixed);
copy.add("Ben");
List<String> view =
    Collections.unmodifiableList(copy);
// view.set(0, "Eve");   → error
copy.set(0, "Eve");
System.out.println(view);
```

Line	fixed	copy	view	Note
`List.of(...)`	[Ann, Dan]	—	—	immutable; rejects `null`
`// fixed.add("Ben")`	[Ann, Dan]	—	—	if uncommented: `UnsupportedOperationException`
`new ArrayList<>(fixed)`	[Ann, Dan]	[Ann, Dan]	—	copied into a new array
`copy.add("Ben")`	[Ann, Dan]	[Ann, Dan, Ben]	—	`fixed` unaffected
`unmodifiableList(copy)`	[Ann, Dan]	[Ann, Dan, Ben]	[Ann, Dan, Ben]	no copy; wraps `copy`
`// view.set(0, "Eve")`	[Ann, Dan]	[Ann, Dan, Ben]	[Ann, Dan, Ben]	if uncommented: `UnsupportedOperationException`
`copy.set(0, "Eve")`	[Ann, Dan]	[Eve, Dan, Ben]	[Eve, Dan, Ben]	the view sees the change
`println(view)`	output: `[Eve, Dan, Ben]`			read-only ≠ immutable

Table 6.5: The three variables after each line.



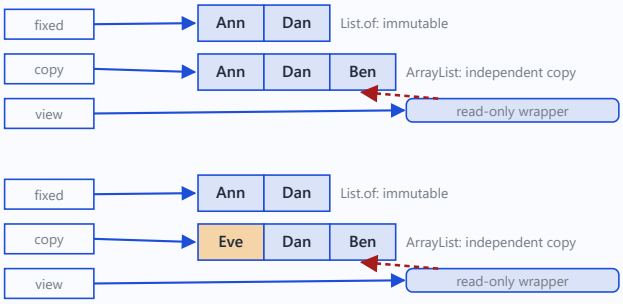


Figure 6.6: Memory. (a) `new ArrayList<>(fixed)` copies the elements into a new array. (b) "Ben" is added to the copy; `fixed` is unchanged. (c) `unmodifiableList` returns a wrapper over `copy`'s array. (d) After `copy.set(0, "Eve")`, the change shows through the view.

The last line settles it: `view` was never modified, yet its contents changed. It is a read-only window onto `copy`'s array. You may not reach through and change anything, but when the room changes, the window shows it.

For a list that truly never changes, use `List.copyOf(copy)`: a new immutable copy unaffected by later changes to `copy`. Table 6.6 compares the three.

Created with	Modifiable itself?	If the source changes?	Cost
`new ArrayList<>(l)`	yes	unaffected (copy)	*O(n)*
`Collections.unmodifiableList(l)`	no	sees the change (view)	*O(1)*
`List.copyOf(l)`	no	unaffected (copy)	*O(n)*

Table 6.6: Copy, read-only view, and immutable list.

Common mistake

Treating "read-only" and "immutable" as synonyms. A class that hands out `Collections.unmodifiableList(list)` stops outside code from changing the list, but its own internal changes show up in what the outside holds. To hand out a frozen snapshot, return `List.copyOf(list)`.

6.3 List methods

Beyond what it inherits from `Collection`, `List` defines many position-based methods. This section covers the common ones and their traps. The examples use `ArrayList`; every `List` class *behaves* the same, only the costs differ.

6.3.1 The main methods and index bounds

Group	Method	Description
Adding	`add(E e)`	appends at the end, returns `true`
	`add(int index, E e)`	inserts at the given index; later elements shift one place right
Removing	`remove(Object o)`	removes the first occurrence; `true` if found
	`remove(int index)`	removes and returns the element at the index
Access	`get(int index)`	returns the element at the index
	`set(int index, E e)`	replaces the element, returns the old one
Length	`size()`	number of elements

Table 6.7: The main methods of the `List` interface.

`E` is the type parameter: `String` for a `List<String>`. Read return values carefully. `set` returns the *old* value: on `[A, B, C]`, `set(0, "X")` returns "A". `remove(int)` returns the removed element, `remove(Object)` a `boolean` saying whether anything was removed.

Index-based methods first check the index and throw `IndexOutOfBoundsException` if it is invalid. The valid range depends on the method:

- `get`, `set`, `remove(int)`: *0 … size() − 1*; *index < 0* or *index ≥ size()* is an error.
- `add(int, E)`: *0 … size()*. `add(size(), e)` appends and is legal; only *index > size()* is an error.

This makes sense: `get` asks for an existing element, while `add` opens a slot. A list of n elements has $n + 1$ gaps: before the first, between each pair, and after the last.

Example 6.2

Let `l = [A, B, C]` (length 3). Which calls are valid?

1. `l.get(3)`: valid indices are 0–2; 3 ≥ 3, so `IndexOutOfBoundsException`.
2. `l.add(3, "D")`: `add`'s bound is `size()`, and 3 ≤ 3, so D is appended: `[A, B, C, D]`.
3. `l.set(0, "X")`: valid; `[X, B, C, D]`, returning "A".
4. `l.add(5, "E")`: *5 > size() = 4*, an error. A list cannot grow by leaving a hole.

6.3.2 Code trace: duplicates, null, and search

Two properties set a list apart from a set: duplicates are allowed and order matters. This program tests both, plus a `null` element:

```
public class ListDemo {   // import java.util.ArrayList; import java.util.List;
    public static void main(String[] args) {
        List<String> names = new ArrayList<>();
        names.add("Ann");
        names.add("Ann");              // duplicate: allowed
        names.add(null);               // null: allowed
        names.add("Ann");
        System.out.println(names);             // [Ann, Ann, null, Ann]
        System.out.println(names.size());      // 4
        System.out.println(names.indexOf("Ann"));        // 0 (first occurrence)
        System.out.println(names.lastIndexOf("Ann"));    // 3
    }
}
```

The slides' version adds `System.out.println(names.indexOf(null));`, showing how the search changes for a `null` target.

Line	Internal array (0–3)	`size`	Output / return
`new ArrayList<>()`	[ ]	0	—
`add("Ann")`	[Ann]	1	`true`
`add("Ann")`	[Ann, Ann]	2	`true` (duplicates allowed)
`add(null)`	[Ann, Ann, null]	3	`true` (`null` allowed)
`add("Ann")`	[Ann, Ann, null, Ann]	4	`true`
`println(names)`	—	4	`[Ann, Ann, null, Ann]`
`size()`	—	4	4
`indexOf("Ann")`	match at cell 0	4	0
`lastIndexOf("Ann")`	from the end: match at cell 3	4	3
`indexOf(null)`	0: no, 1: no, 2: yes	4	2

Table 6.8: Trace of `ListDemo`. `data` is the `ArrayList`'s internal array.

The first line comes from `toString()`: elements in order, comma-separated, in brackets, `null` printed as "null". `size()` gives 4, the element count, not the internal array's capacity of 10 (an empty `ArrayList` allocates 10 cells on the first add).

`indexOf` scans from the front and stops at the first match, comparing with `equals`, not `==`: `"Ann".equals(data[0])` is true, so it returns 0. `lastIndexOf` scans from the back and matches at once in cell 3. Both return −1 if the element is absent, and both cost *O(n)* in the worst case.

The last line is subtle. For a `null` target, `o.equals(data[i])` would throw a `NullPointerException`, so `ArrayList` has two loops: `data[i] == null` for a `null` target, `o.equals(data[i])` otherwise. Cells 0 and 1 hold "Ann", cell 2 `null`, so the call returns 2.

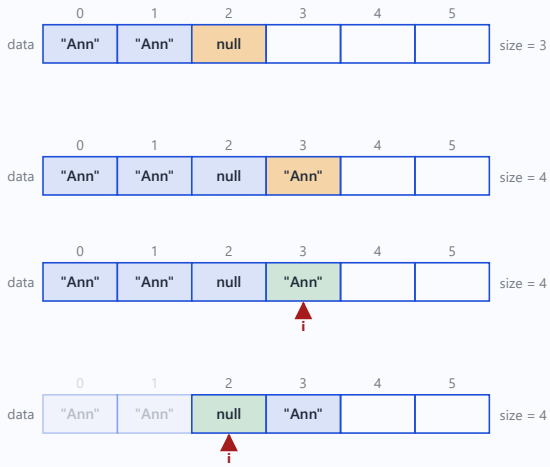


Figure 6.7: The internal array at four moments. (a) `null` added in cell 2. (b) The third "Ann" in cell 3; `size = 4`. (c) `lastIndexOf` starts at the end and stops at cell 3. (d) `indexOf(null)` finds cell 2 with `==`.

The list holds "Ann" three times, each at its own position. If each value may appear only once, use a `Set`.

> **Common mistake**
>
> Assuming every list accepts `null`. `ArrayList` and `LinkedList` do, but a `List.of(...)` list throws `NullPointerException` on a `null` element, and even its `contains(null)` or `indexOf(null)` may throw.

6.3.3 remove(int) or remove(Object)?

`List` has two `remove` methods, one taking an index (`remove(int index)`), the other an element (`remove(Object o)`). Such same-named methods with different parameters are **overloaded**; the compiler picks one by the argument's type.

With `String`s there is no confusion: `remove(1)` is by index, `remove("Ann")` by element. With `Integer`s an integer can be either.

```
List<Integer> nums = new ArrayList<>(List.of(10, 20, 30));
nums.remove(1);                    // removes index 1 → [10, 30]
nums.remove(Integer.valueOf(10));  // removes the value 10 → [30]
nums.remove(Integer.valueOf(99));  // absent → false, list unchanged
nums.remove(5);                    // IndexOutOfBoundsException
```

Method	What it does	Returns	Cost in an array-based list
`remove(int index)`	removes the element at the given position	the removed element (`E`)	*O(n)*: later elements shift
`remove(Object o)`	finds the first occurrence and removes it	whether it removed (`boolean`)	*O(n)*: search + shift

Table 6.9: The two `remove` methods compared.

The slides' version stores the return values:

```
List<Integer> nums =
    new ArrayList<>(List.of(10, 20, 30));
Integer a = nums.remove(1);
boolean b = nums.remove(Integer.valueOf(10));
boolean c = nums.remove(Integer.valueOf(99));
nums.remove(5);   // IndexOutOfBoundsException
```

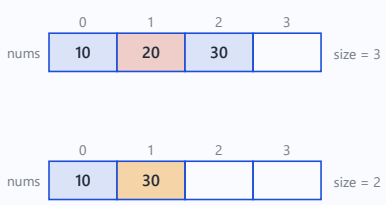
Call	Method chosen	List (after)	Returns
(start)	—	[10, 20, 30]	—
`remove(1)`	`remove(int)` : index 1	[10, 30]	`a = 20`
`remove(Integer.valueOf(10))`	`remove(Object)` : value 10	[30]	`b = true`
`remove(Integer.valueOf(99))`	`remove(Object)` : value 99	[30]	`c = false`
`remove(5)`	`remove(int)` : index 5	[30]	`IndexOutOfBoundsException`

Table 6.10: Trace of the `remove` calls.

Java's overload rule first looks for a match needing no boxing or unboxing. `remove(1)` passes a primitive `int` , which fits `remove(int)` exactly; `remove(Object)` would need boxing. So 20 at index 1 is removed, 30 shifts left, and 20 is returned.

`Integer.valueOf(10)` is already an object, so `remove(Object)` fits directly. The scan finds `Integer.valueOf(10).equals(10)` true at index 0, removes it, and returns `true` . For 99 the scan reaches the end; the list is unchanged and the result is `false` .

At the last call the list has one element. `remove(5)` checks *5 < size() = 1*, fails, and Java reports `Index 5 out of bounds for length 1` .



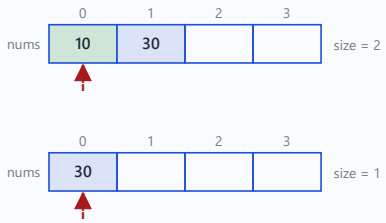


Figure 6.8: (a) `remove(1)`: cell 1 is to go. (b) 30 shifted left; `a = 20`. (c) `remove(Integer.valueOf(10))` finds the value in cell 0. (d) The search for 99 reached the end with no match.

Common mistake

Writing `l.remove(7)` to remove the *value* 7 from a `List<Integer>`. It removes index 7 and throws if the list is short. Pass an object: `l.remove(Integer.valueOf(7))` or `l.remove((Integer) 7)`. Conversely, `l.remove(l.get(1))` removes by *value*, since `get` returns an `Integer`.

6.3.4 Bulk operations with lambdas: replaceAll and sort

Applying one operation to every element is common. An indexed loop works, but since Java 8 `List` has methods that take the work as a **lambda expression**, a short unnamed function: `name -> name.toUpperCase()` means "take a `name`, return it in upper case."

```
List<String> names =
        new ArrayList<>();
names.add("Tim");
names.add("Tom");
names.add("Bob");
names.replaceAll(
        name -> name.toUpperCase());
System.out.println(names);
// [TIM, TOM, BOB]
```

`replaceAll` calls the lambda for each index and writes the result back *in place*; no new list is created. The result is `[TIM, TOM, BOB]`.

```
List<Integer> nums =
        new ArrayList<>();
nums.add(5);
nums.add(3);
nums.add(1);
nums.sort((s1, s2) ->
        Integer.compare(s1, s2));
System.out.println(nums);
// [1, 3, 5]
```

`sort` takes a **comparator**, a lambda returning a negative number, zero, or a positive number as the first element is smaller than, equal to, or larger than the second, exactly what `Integer.compare(s1, s2)` does. For natural order, `nums.sort(null)` or `nums.sort(Comparator.naturalOrder())` is shorter; for reverse order, pass `Comparator.reverseOrder()`.

The slides combine both jobs in one program, with one fix to the upper-casing:

```
List<String> names = new ArrayList<>(
    List.of("Tim", "Tom", "Bob"));
names.replaceAll(
    name -> name.toUpperCase(Locale.ROOT));
List<Integer> nums = new ArrayList<>(
    List.of(5, 3, 1));
nums.sort((s1, s2) -> Integer.compare(s1, s2));
System.out.println(names + " " + nums);
```

`Locale.ROOT` matters on any machine with Turkish settings. Without arguments, `toUpperCase()` uses the computer's locale, and in the Turkish locale a dotted "i" becomes "İ": "Tim" turns into "TİM". The locale-independent `Locale.ROOT` gives "TIM" everywhere.

Java sorts object lists with **TimSort**, which first looks for stretches already in ascending or descending order (*runs*). On `[5, 3, 1]`:

Call	Result	Meaning
`compare(3, 5)`	−1	3 < 5: the array starts with a descending run
`compare(1, 3)`	−1	1 < 3: the descending run continues; length 3
(reversal)	—	the whole array is one descending run; it is reversed: `[1, 3, 5]`

Table 6.11: The comparisons `sort` makes on `[5, 3, 1]`.

Two comparisons finish the sort. On an already sorted or exactly reversed list TimSort makes $n - 1$ comparisons; in general it is $O(n \log n)$. The program prints `[TIM, TOM, BOB] [1, 3, 5]`.



Figure 6.9: (a) `replaceAll` writes "TOM" into cell 1. (b) `[5, 3, 1]` before sorting. (c) After two comparisons, all three elements form one descending run. (d) The run reversed: `[1, 3, 5]`.

Common mistake

Writing a comparator as `(a, b) -> a - b`. Fine for small numbers, but with a very large `a` and a very negative `b` the subtraction overflows `int`, the sign flips, and the sort silently goes wrong. `Integer.compare(a, b)` never overflows.

6.3.5 Indexed access, generics, and boxing

`get(i)` reaches an element by position, and it is tied up with two features of Java: generics and boxing.

```
List<String> names =
        new ArrayList<>();
names.add("Ann");
names.add("Tom");
names.add("Bob");
String firstName = names.get(0);
System.out.println(firstName);  // Ann
names.get(3);  // IndexOutOfBounds
```

```
List<String> names =
        new ArrayList<>();
names.add("Ann");
names.add(5);   // compile error

List<Integer> nums =
        new ArrayList<>();
nums.add(5);   // autoboxing
int x = nums.get(0);  // unboxing
```

With **generics**, a class or interface declares the element type through a type parameter. Given `List<String>`, the compiler admits only `String`s, so `names.add(5)` is a *compile-time* error, caught before the program runs, and `get` returns a `String` with no cast.

A type parameter cannot be primitive: instead of `List<int>` we write `List<Integer>`, using `int`'s **wrapper class**. Every primitive has one (`Double`, `Character`, `Boolean`, ...), and the compiler converts automatically:

- **Autoboxing**: `nums.add(5)` becomes `nums.add(Integer.valueOf(5))`; the list holds a reference to an `Integer` object containing 5.
- **Unboxing**: `int x = nums.get(0);` becomes `nums.get(0).intValue()`.

The slides trace all these steps:

```
List<String> names = new ArrayList<>();
names.add("Ann");
names.add("Tom");
names.add("Bob");
String firstName = names.get(0);
System.out.println(firstName);
List<Integer> nums = new ArrayList<>();
nums.add(5);          // Integer.valueOf(5)
int x = nums.get(0);  // .intValue()
names.get(3);            // IndexOutOfBounds
```

Line	What the compiler produces	State	Result
`new ArrayList<>()`	—	`names` empty	accepts only `String`
three `add`s	—	[Ann, Tom, Bob]	indices 0, 1, 2
`get(0)`	—	check *0 < 3* passes	"Ann", *O(1)*
`println(firstName)`	—	—	output: `Ann`
`nums.add(5)`	`add(Integer.valueOf(5))`	an `Integer` reference in the list	boxing
`int x = nums.get(0)`	`get(0).intValue()`	`x = 5`	unboxing

Line	What the compiler produces	State	Result
`names.get(3)`	—	*3* < *3* false	`IndexOutOfBoundsExceptio n`

Table 6.12: Trace of `get`, generics, and boxing.

`get(0)` on an `ArrayList` checks the index and reads `data[0]` directly. An address calculation finds the cell whatever the list's length, so `get` is *O(1)*. On a `LinkedList` it must walk nodes and is expensive (Section 6.4.2).

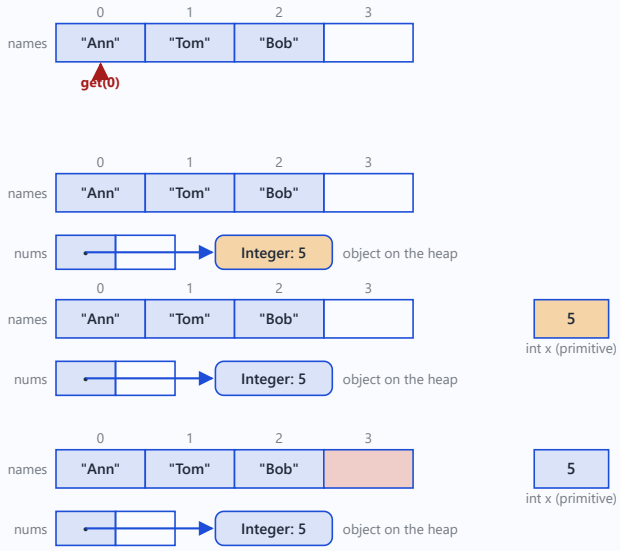


Figure 6.10: (a) `get(0)` reads cell 0 directly. (b) `nums.add(5)`: the list holds a reference to an `Integer` on the heap. (c) `get(0).intValue()` gives the primitive 5 to `x`. (d) `get(3)`: valid indices are 0–2, so it throws.

Boxing has a hidden cost. An `int` takes 4 bytes; an `Integer` object with its header about 16, plus the list's reference to it. With millions of numbers, an `int[]` beats a `List<Integer>` in both memory and speed.

Common mistake

Assuming unboxing is always safe. If a `List<Integer>` holds `null`, `int x = l.get(0);` calls `null.intValue()` and throws `NullPointerException`, a surprise on a line with no visible method call.

6.3.6 Spliterator: a splittable iterator

An iterator delivers elements one at a time, front to back. To process a large list on a multicore processor, we want to split the work and give each part its own **thread**. `Spliterator` ("split" + "iterator") grew out of this need.

```
List<String> words = new ArrayList<>();
words.add("Java");
words.add("Python");
words.add("C++");
Spliterator<String> split =
        words.spliterator();
while (split.tryAdvance(
        System.out::println)) {
    // each call processes one element
}
```

`Spliterator` has two basic methods:

- `tryAdvance(action)`: if an element remains, applies the action and returns `true`; otherwise returns `false`. It merges `hasNext()` and `next()`. Here the action is the method reference `System.out::println`; the loop body is empty because the work happens in the condition. Output: Java, Python, C++, one per line.
- `trySplit()`: hands part of the remaining elements to a new `Spliterator` and returns it, so two threads can take the two parts; returns `null` if too few remain.

Parallel streams (`parallelStream()`) use exactly this: `trySplit` to divide, `tryAdvance` to process each part. The slides trace a split:

```
List<String> words = new ArrayList<>(
    List.of("Java", "Python", "C++", "Go"));
Spliterator<String> a = words.spliterator();
Spliterator<String> b = a.trySplit();
while (b.tryAdvance(System.out::println)) { }
while (a.tryAdvance(System.out::println)) { }
```

`ArrayList`'s spliterator holds two numbers: the next `index` and the end bound, `fence`. The fence starts at −1, "not yet fixed," and is set from `size()` on first use, so elements added before then are still covered.

Call	`a`	`b`	**Returns / output**
`words.spliterator()`	index 0, fence −1	—	no fence yet
`a.trySplit()`	*[2, 4)*	*[0, 2)*	fence = 4, mid = *(0 + 4) / 2 = 2*
`b.tryAdvance`	*[2, 4)*	*[1, 2)*	Java; `true`
`b.tryAdvance`	*[2, 4)*	*[2, 2)*	Python; `true`
`b.tryAdvance`	*[2, 4)*	*[2, 2)*	no element; `false`
`a.tryAdvance`	*[3, 4)*	—	C++; `true`
`a.tryAdvance`	*[4, 4)*	—	Go; `true`

Call	`a`	`b`	Returns / output
`a.tryAdvance`	*[4, 4)*	—	no element; `false`

Table 6.13: Trace of `trySplit` and `tryAdvance`. Ranges are written as *[index, fence)*.

Note that the *first* half goes to the new spliterator `b` and the old one `a` keeps the second. The midpoint is *(index + fence) / 2* with integer division; for five elements, *(0 + 5) / 2 = 2*, so the new part gets 2 and the old 3. Processed one after the other, the parts print Java, Python, C++, Go; a parallel stream could run them on two cores at once.

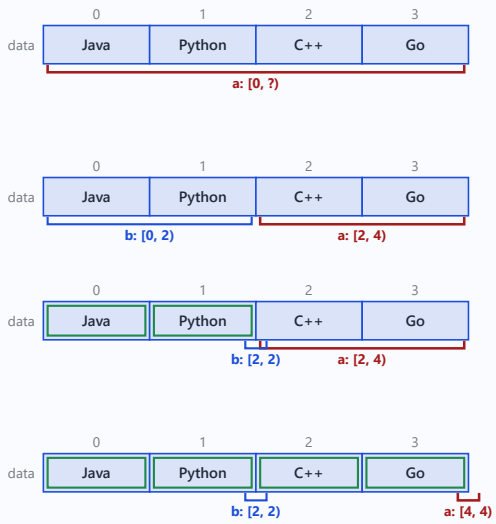


Figure 6.11: (a) `spliterator()` : fence not yet fixed. (b) `trySplit()` : `b` is *[0, 2)*, `a` *[2, 4)*. (c) `b` processed its two elements. (d) `a` is done too; each element was processed once.

6.4 Traversal

Visiting every element in order is **traversal**: summing, finding the largest, printing, removing matches. It looks simple, but with the wrong tool a linear loop turns quadratic or the program stops with an exception.

6.4.1 Three ways to traverse a collection

There are three ways. All yield the same elements in the same order; they differ in what they allow and what they cost.

1. Indexed for loop.

```
int n = list.size();
for (int i = 0;
     i < n; i++) {
  String s =
      list.get(i);
  ...
}
```

Only for a `List`, since it needs `get(i)`; natural when the loop needs the index, say to print "item 3" or compare neighbors.

2. forEach or for-each.

```
list.forEach(
  System.out::println);

// or for-each:
for (String s : list) ...
```

Works for any `Iterable` and is shortest. `list.forEach(...)` takes a lambda or method reference; the for-each loop compiles to an iterator.

3. An explicit iterator.

```
Iterator<String> it =
    list.iterator();
while (it.hasNext()) {
  String s = it.next();
}
```

Longest, but it alone lets you remove elements safely mid-traversal, for reasons explained in Section 6.4.4.

Cost depends on the list. On an `ArrayList`, `get(i)` is *O(1)* and all three take *O(n)*. On a `LinkedList`, `get(i)` walks the nodes in *O(n)*, and the indexed loop repeats the walk at every step: $O(n^2)$, against *O(n)* for forEach and the iterator.

6.4.2 Code trace: the cost of an indexed loop over a LinkedList

`LinkedList` is doubly linked: each node knows its successor and predecessor, and the list keeps `first` and `last`. `get(i)` walks to the node with the internal `node(i)`, forward from the front if $i < n/2$, else backward from the end, so it walks $min(i, n - 1 - i)$ links.

This program totals the links walked by all `get` calls in `steps`:

```
List<String> list = new LinkedList<>(
    List.of("A", "B", "C", "D", "E", "F"));
int n = list.size(), steps = 0;
for (int i = 0; i < n; i++) {
  String s = list.get(i);  // node(i) walks
  steps += Math.min(i, n - 1 - i);
}
System.out.println(steps);
```

`i`	Decision	Start	Links walked	`steps` (total)
0	$0 < 3$	`first`	0	0
1	$1 < 3$	`first`	1	1
2	$2 < 3$	`first`	2	3
3	$3 \geq 3$	`last`	2	5
4	$4 \geq 3$	`last`	1	6
5	$5 \geq 3$	`last`	0	6

Table 6.14: Links walked by the indexed loop over a 6-node `LinkedList` ($n/2 = 3$).

It prints 6, but what matters is growth. For even n, the walks give $0, 1, \ldots, n/2 - 1$ twice, once from each end:

> $2 \cdot (0 + 1 + \ldots + (n/2 - 1)) = 2 \cdot (n/2)(n/2 - 1)/2 = n^2/4 - n/2$

For $n = 6$ this is $9 - 3 = 6$, matching the table; for $n = 1000$ it is 249,500. The total grows like $n^2/4$: the loop is $O(n^2)$. Starting from the nearer end only halves the work.

An iterator remembers its node and advances one link per `next()`: $n - 1 = 5$ links for all six elements, $O(n)$.

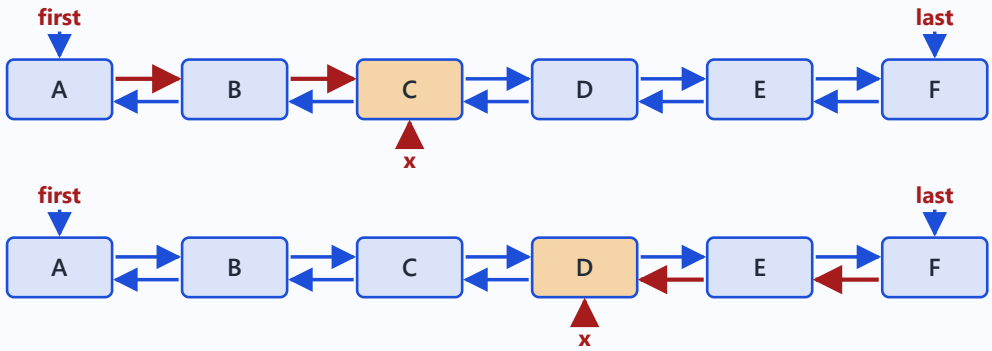


Figure 6.12: How `get` walks a `LinkedList`. Top, `get(2)`: $2 < 3$, so two links forward from `first`. Bottom, `get(3)`: $3 \geq 3$, so two links back from `last`. Highlighted arrows are the links walked.

Common mistake

Writing `for (int i = 0; i < l.size(); i++) … l.get(i) …` in a method taking a `List`. Fast when tested with an `ArrayList`, quadratic the day it gets a `LinkedList`. For a list of unknown type, use for-each or an iterator.

6.4.3 The Iterator interface and the cursor

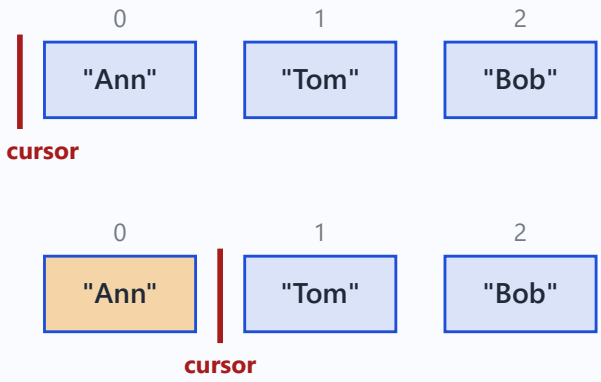
`Iterator` defines three methods:

`hasNext()`
: Any elements left? Returns `false` at the end. It changes no state, so repeated calls give the same answer.

`next()`
: Returns the next element and advances; throws `NoSuchElementException` if none is left.

`remove()`
: Removes the element returned by the last `next()`. Optional: collections that do not support it throw `UnsupportedOperationException`.

Picture a **cursor** that, like a word processor's, sits *between* elements. It starts before the first; each `next()` jumps over one element and returns it. Past the last element, `hasNext()` is false.



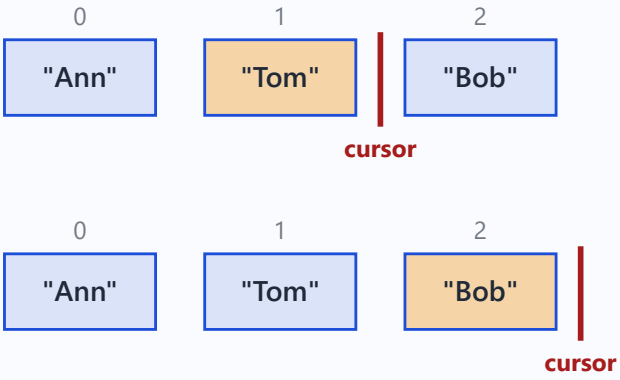


Figure 6.13: The cursor model. (a) After `iterator()` the cursor precedes the first element; `hasNext()` is true. (b) `next()` returns "Ann" and the cursor moves past it. (c) The second `next()` returns "Tom". (d) The third returns "Bob"; the cursor is at the end and `hasNext()` is false.

An iterator is separate from the list; several can be open at once, each with its own cursor. In `ArrayList`'s iterator the cursor is an integer, `cursor`, the index the next `next()` will return, and `hasNext()` just checks `cursor != size`.

```
Iterator<String> iterator = names.iterator();
while (iterator.hasNext()) {
    String name = iterator.next();
    System.out.println(name);
}
```

Call	Check	Returns	`cursor` (after)	Output
`iterator()`	—	iterator	0	—
`hasNext()`	0 ≠ 3	`true`	0	—
`next()`	—	"Ann"	1	Ann
`hasNext()`	1 ≠ 3	`true`	1	—
`next()`	—	"Tom"	2	Tom
`hasNext()`	2 ≠ 3	`true`	2	—
`next()`	—	"Bob"	3	Bob
`hasNext()`	3 = 3	`false`	3	loop ends

Table 6.15: Trace of an iterator over `[Ann, Tom, Bob]` (`size = 3`).

One more `next()` after the loop finds nothing ahead of the cursor and throws `NoSuchElementException`; hence every `next()` is guarded by `hasNext()`, or the loop is a for-each.

Common mistake

Calling `next()` twice in one loop body: `while (it.hasNext()) { if (it.next().startsWith("A")) System.out.println(it.next()); }`. Each call advances the cursor, so the second returns the *following* element and may throw on the last round. Store the element first: `String s = it.next();`.

6.4.4 Removing while traversing: ConcurrentModificationException

Removing matching elements during traversal is a common need, and the natural-looking code is wrong:

```
for (String s : names)
    if (s.startsWith("A"))
        names.remove(s);
// ConcurrentModification-
// Exception (fail-fast)
```

Correct code uses the iterator's own `remove()` or `removeIf`:

```
Iterator<String> it =
        names.iterator();
while (it.hasNext())
    if (it.next().startsWith("A"))
        it.remove();
// or, briefly:
names.removeIf(s -> s.startsWith("A"));
```

The wrong version fails because of a counter. Every `ArrayList` has a field `modCount`, incremented by every **structural modification**, any add or remove that changes the element count (`set` does not count). A new iterator saves the counter as `expectedModCount`, and each `next()` compares the two. If they differ, the list changed behind the iterator's back, and it throws `ConcurrentModificationException`.

Definition 6.3

A **fail-fast** iterator detects that its collection was structurally modified from outside during traversal and throws at once, making the error visible instead of silently producing wrong results.

The slides write out what the compiler makes of the for-each loop:

```
List<String> names = new ArrayList<>();
names.add("Ann"); names.add("Ben");
names.add("Alice");              // modCount = 3
// for (String s : names) ... compiler:
Iterator<String> it = names.iterator();
while (it.hasNext()) {
  String s = it.next();
  if (s.startsWith("A"))
    names.remove(s);             // bypasses it!
}
```

Step	List	size	cursor	modCount / exp	Note
three add s	[Ann, Ben, Alice]	3	—	3 / —	each add increments the counter
iterator()	[Ann, Ben, Alice]	3	0	3 / 3	counter saved
hasNext()	[Ann, Ben, Alice]	3	0	3 / 3	0 ≠ 3 → true
next()	[Ann, Ben, Alice]	3	1	3 / 3 ✓	returns "Ann"
names.remove(s)	[Ben, Alice]	2	1	4 / 3	iterator unaware
hasNext()	[Ben, Alice]	2	1	4 / 3	1 ≠ 2 → true
next()	[Ben, Alice]	2	1	4 ≠ 3 ✗	ConcurrentModificationException

Table 6.16: Trace of removing from outside. exp is the iterator's expectedModCount field.

The fifth row is the root of the problem. Removing "Ann" shifts "Ben" to index 0, but cursor is still 1; unchecked, the next next() would return "Alice" and "Ben" would be skipped. The fail-fast check (modCount 4, expectedModCount 3) prevents this silent skip.

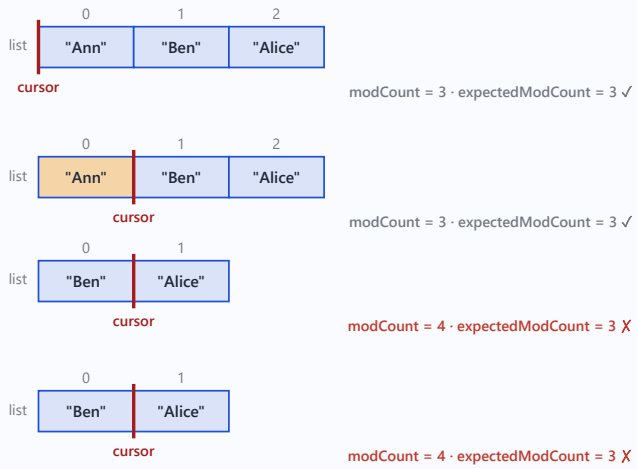


Figure 6.14: (a) iterator() : cursor at the start, counters equal. (b) next() returned "Ann". (c) names.remove(s) : elements shifted, modCount = 4 ; "Ben" is behind the cursor. (d) next() sees the counters differ and stops.

Fail-fast is a safeguard, not a guarantee: the check runs only in `next()`. If the removal comes near the end, `hasNext()` may return false first. Removing "Ann" from outside while for-each-ing over `[Ben, Ann, Eve]` leaves `cursor = 2`, `size = 2`, and the loop ends quietly with `[Ben, Eve]`, "Eve" never visited. The Java documentation therefore calls fail-fast behavior a debugging aid only.

6.4.5 Code trace: safe removal with it.remove()

```
Iterator<String> it = names.iterator();
while (it.hasNext()) {
  String s = it.next();
  if (s.startsWith("A"))
    it.remove();   // keeps the counters equal
}
System.out.println(names);
```

The iterator's `remove()` removes the element and adjusts the iterator's state. Besides `cursor` it keeps `lastRet`, the index of the last element returned, or −1 if there is none or it was already removed.

`it.remove()` then:

1. If `lastRet = -1`, throws `IllegalStateException`.
2. Removes the element at `lastRet`; `modCount` goes up.
3. Sets `cursor = lastRet`, stepping back so that the element shifted into the gap is not skipped.
4. Sets `lastRet = -1`: no second `remove()` without a `next()`.
5. Sets `expectedModCount = modCount`, since the iterator made the change itself.

Step	List	`cursor`	`lastRet`	`modCount / exp`	Note
`iterator()`	[Ann, Ben, Alice]	0	−1	3 / 3	—
`next()`	[Ann, Ben, Alice]	1	0	3 / 3	"Ann"; starts with A
`it.remove()`	[Ben, Alice]	0	−1	4 / 4	cursor stepped back
`next()`	[Ben, Alice]	1	0	4 / 4	"Ben"; kept
`next()`	[Ben, Alice]	2	1	4 / 4	"Alice"; starts with A
`it.remove()`	[Ben]	1	−1	5 / 5	—
`hasNext()`	[Ben]	1	−1	5 / 5	1 = 1 → `false`; no error

Table 6.17: Trace of safe removal with `it.remove()`. Start: `[Ann, Ben, Alice]`, `modCount = 3`.

It prints `[Ben]`: both A-names are gone, and "Ben" was not skipped. The critical step is `cursor = 0` in the third row; removing "Ann" moved "Ben" to index 0, and without stepping back "Ben" would be skipped.

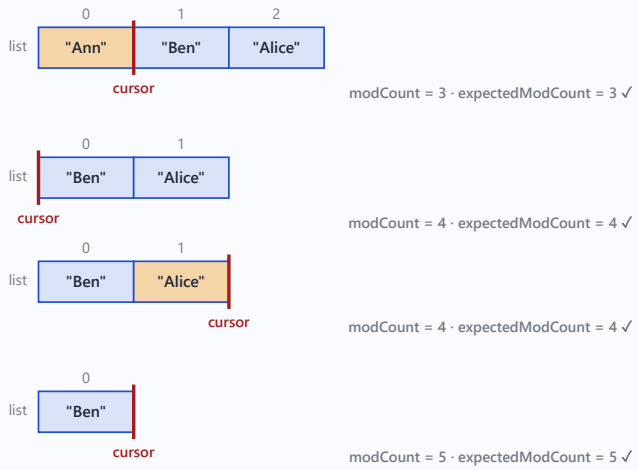


Figure 6.15: (a) `next()` returned "Ann". (b) `it.remove()`: "Ann" removed, cursor back at the start, counters both 4. (c) `next()` returned "Alice". (d) After the second `it.remove()`: `[Ben]`, counters both 5.

`removeIf` does it in one line: `names.removeIf(s -> s.startsWith("A"));`. On an `ArrayList` it marks the doomed elements and compacts the rest in one pass, $O(n)$; removing one at a time through an iterator can shift $O(n^2)$ times.

> **Common mistake**
>
> Calling `it.remove()` twice in a row, or before any `next()`. It removes only the element the last `next()` returned, and only once; a second call finds `lastRet = -1` and throws `IllegalStateException`.

6.4.6 The ListIterator interface

`Iterator` goes one way and only removes. On a `List`, `list.listIterator()` gives a `ListIterator`, which moves both ways and can add, remove, and replace along the way. Its cursor, too, sits between elements.

Method	Purpose
`hasNext()` / `next()`	any element ahead? / return the next element, move the cursor forward
`hasPrevious()` / `previous()`	any element behind? / return the previous element, move the cursor back
`nextIndex()` / `previousIndex()`	index of the element `next()` / `previous()` would return (initially 0 / −1)
`add(e)`	inserts the element before the cursor (the next `next()` is unaffected)
`remove()`	removes the element returned by the last `next()` / `previous()`
`set(e)`	replaces the element returned by the last `next()` / `previous()`

Table 6.18: `ListIterator` methods.

One principle governs: `remove()` and `set(e)` act on "the last element returned," while `add(e)` inserts at the cursor. Calling `remove` or `set` right after `add` or `remove`, with no `next` / `previous` between, throws `IllegalStateException`.

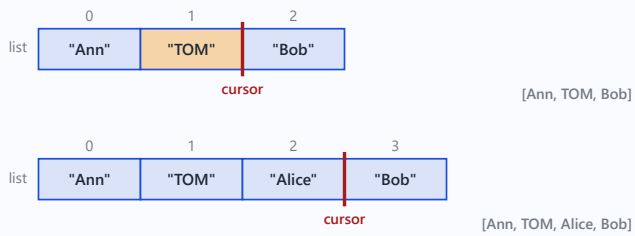
Watch a `next()` right after a `previous()`: the first jumps back over an element, the second forward over the *same* one, returning it again. With `listIterator(1)` on `[A, B, C]`, `previous()` and then `next()` both return A.

```
ListIterator<String> it = names.listIterator();
it.next();          // "Ann"
it.next();          // "Tom"
it.set("TOM");     // [Ann, TOM, Bob]
it.add("Alice");    // [Ann, TOM, Alice, Bob]
it.previous();      // "Alice"
it.remove();        // [Ann, TOM, Bob]
```

Call	List and cursor	Returns	`lastRet`
`listIterator()`	| Ann Tom Bob	—	−1
`next()`	Ann | Tom Bob	"Ann"	0
`next()`	Ann Tom | Bob	"Tom"	1
`set("TOM")`	Ann TOM | Bob	—	1
`add("Alice")`	Ann TOM Alice | Bob	—	−1
`previous()`	Ann TOM | Alice Bob	"Alice"	2
`remove()`	Ann TOM | Bob	—	−1

Table 6.19: Trace of a `ListIterator`. In the List column, `|` is the cursor. Start: `[Ann, Tom, Bob]`.

`set("TOM")` overwrites the last returned "Tom" without moving the cursor. `add("Alice")` inserts before the cursor and moves past it, so a `next()` would still return "Bob"; after `add`, `lastRet` is −1 and a `set` would throw. `previous()` returns "Alice", and `remove()` removes it. Result: `[Ann, TOM, Bob]`, cursor between TOM and Bob (`nextIndex() = 2`).



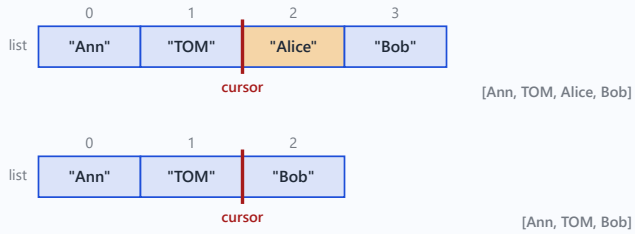


Figure 6.16: Orange marks the last returned element. (a) `set("TOM")` : that element changed; the cursor stayed. (b) `add("Alice")` : inserted before the cursor; no last returned element. (c) `previous()` returned "Alice". (d) `remove()` removed it.

`ListIterator` 's greatest value is cost. `LinkedList.add(i, e)` first walks to node i, *O(n)*, but a `ListIterator` already stands by a node, so its `add` , `remove` , and `set` relink in *O(1)*. Algorithms that sweep a list once, inserting and removing on the way, exploit `LinkedList` only this way.

6.4.7 Code trace: moving forward and backward

```
ListIterator<String> iterator =
        names.listIterator();
// traverse the list forward
while (iterator.hasNext()) {
    String name = iterator.next();
    System.out.println(name);
}
// the cursor is now at the end: traverse backward
while (iterator.hasPrevious()) {
    String name = iterator.previous();
    System.out.println(name);
}
```

The program prints, one per line:

```
Ann
Tom
Bob
Bob
Tom
Ann
```

Both loops share *one* iterator. After the first, the cursor is at the end (`cursor = 3`); `previous()` then returns "Bob", "Tom", "Ann" and brings it back to 0, where `hasPrevious()` (*cursor > 0*) fails. *3 + 3 = 6* lines in all.

Call	Check	Returns / prints	`cursor` (after)
`next()` ×3	0, 1, 2 < 3	Ann, Tom, Bob	1, 2, 3
`hasNext()`	3 < 3 fails	`false`	3
`previous()` ×3	3, 2, 1 > 0	Bob, Tom, Ann	2, 1, 0
`hasPrevious()`	0 > 0 fails	`false`	0

Table 6.20: How `cursor` changes in forward and backward traversal.

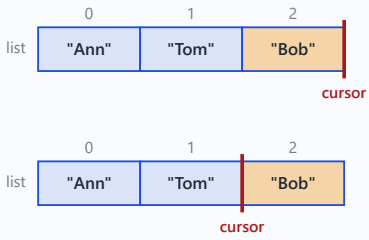


Figure 6.17: (a) End of forward traversal: "Bob" returned, cursor at the end. (b) First backward step: `previous()` returns "Bob" again, cursor back one.

To start from the end, skip the first loop: `names.listIterator(names.size())` puts the cursor after the last element.

6.5 ArrayList

Having seen what `List` does, we now open up *how* its classes do it, starting with `ArrayList`, the most used list class in Java.

6.5.1 Classes that implement List

Six `List` structures stand out in the standard library. They share methods but differ inside, and so in strengths and weaknesses.

`ArrayList`
: A growable array; fast indexed access; the default choice.

`LinkedList`
: A doubly linked list; fast at the ends, slow by index.

`Vector`
: An old `ArrayList` with `synchronized` methods.

`Stack`
: A stack derived from `Vector`: `push`, `pop`, `peek` (last in, first out).

`CopyOnWriteArrayList`

Copies its array on every write; suited to many reader threads.

`Arrays.asList()`

Turns an array into a fixed-size `List` view.

6.5.2 A growable array

An array's length is fixed at creation, yet a list's eventual size is usually unknown. `ArrayList` keeps its elements in an ordinary array and hides that array's length. The length is the **capacity**, the element count the **size** (`size`); always *capacity ≥ size*, the spare cells awaiting future adds.

When the array is full and another add arrives, `ArrayList` allocates an array about 1.5 times larger, copies everything over, and drops the old one. The copy is *O(n)* but rare, since many adds fit before the next one; appending is therefore **amortized *O(1)***.

`ArrayList` never shrinks by itself: add a million elements and remove them, and the capacity stays a million. `trimToSize()` cuts it to the current size.

Two details of real Java: `new ArrayList<>()` allocates nothing until the first add, which creates 10 cells; and the new capacity is *old + old/2*, coded as `old + (old >> 1)`, since a one-bit right shift halves an integer. The slides' trace starts at capacity 4 to keep the steps few.

6.5.3 Code trace: add(e) and growth

The heart of `ArrayList` 's append:

```
public boolean add(E e) {
  if (size == data.length) {
    int newCap = Math.max(data.length + 1, data.length + data.length / 2);
    data = Arrays.copyOf(data, newCap);
  }
  data[size++] = e;
  return true;
}
```

If the array is full (`size == data.length`), the method computes a new capacity — `Math.max` ensures growth even at capacity 0 or 1, where *1 + 1/2 = 1* — and `Arrays.copyOf` creates the bigger array, copying the old elements to its front and leaving the rest `null` . `data` then points to the new array, and the unreferenced old one is garbage-collected. Finally `data[size++] = e` fills the first empty cell and increments the size.

Append D, E, F, G to `[A, B, C]` (capacity 4, size 3):

Call	Full?	Growth	Copies	Size	Capacity
`add(D)`	3 == 4? no	—	0	4	4
`add(E)`	4 == 4? **yes**	*4 + 4/2 = 6*	4	5	6
`add(F)`	5 == 6? no	—	0	6	6
`add(G)`	6 == 6? **yes**	*6 + 6/2 = 9*	6	7	9

Table 6.21: Trace of `add(e)` and growth. Start: `[A, B, C]`, capacity 4.

Two adds are cheap single writes; two first copy everything, *4 + 6 = 10* copies in all. Growth is by a ratio — 4, 6, 9, then 13, 19 — so the cheap adds between growths keep increasing and growths grow rarer.

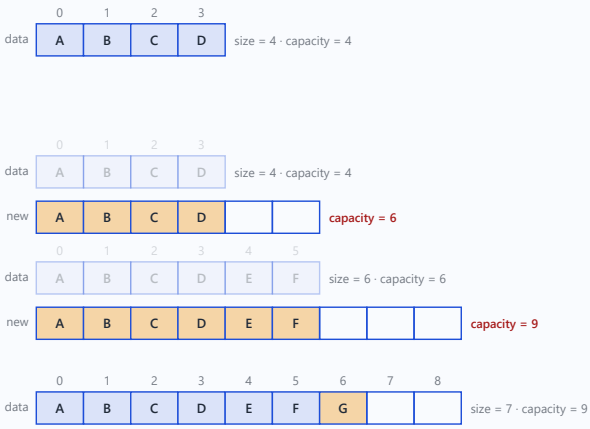


Figure 6.18: (a) At `add(E)` the array is full: size 4, capacity 4. (b) A new 6-cell array holds copies of the 4 elements; the old one is faded. (c) Second growth at `add(G)` : 9 cells, 6 copies. (d) G written: size 7, capacity 9.

6.5.4 Amortized analysis: why grow by doubling?

An add that grows the array takes *O(n)*. Appending counts as *O(1)* because we look at a long sequence of operations, not one.

Definition 6.4

The **amortized cost** of an operation is the total cost of a long sequence divided by its length. No probability is involved: it is an upper bound for every possible sequence. Finding one is **amortized analysis**.

The growth rule decides everything. Figure 6.19 counts the copies actually made over 40 adds, starting at capacity 4, when the capacity doubles, grows by 1.5, or grows by a fixed 4.

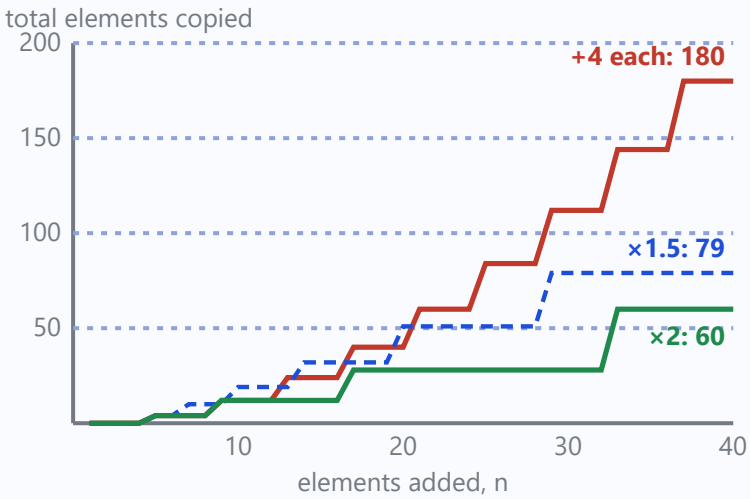


Figure 6.19: Accumulated copies over 40 adds from capacity 4: doubling 60, ×1.5 79, +4 at a time 180. The fixed-increment curve keeps steepening.

The chart's numbers can be checked by hand.

Example 6.3

Forty elements go into an empty dynamic array of capacity 4. How many copies does each rule make?

1. **Doubling (×2).** 4 → 8 → 16 → 32 → 64, growing at 4, 8, 16, 32 elements: *4 + 8 + 16 + 32 = 60*.
2. **×1.5.** 4 → 6 → 9 → 13 → 19 → 28 → 42 (*old + old/2*, integer division): *4 + 6 + 9 + 13 + 19 + 28 = 79*.
3. **+4.** 4 → 8 → … → 40, growing at 4, 8, …, 36: *4 + 8 + … + 36 = 4 · (1 + 2 + … + 9) = 4 · 45 = 180*.

At 40 elements the gap is threefold, and it widens fast. A general calculation shows why.

Theorem 6.1 (Amortized cost of geometric growth)

Adding n elements to a dynamic array that doubles when full makes fewer than $2n$ copies; growing by 1.5 makes at most about $3n$. Either way appending is amortized $O(1)$.

Proof. With doubling the copy counts are $c, 2c, 4c, \ldots, K$, where K, the last, is below n because the array held K elements then. Read backward they halve, so the sum is $K \cdot (1 + 1/2 + 1/4 + \ldots) < 2K < 2n$.

With 1.5, each term is about $2/3$ of the next, so the sum is about $K \cdot (1 + 2/3 + 4/9 + \ldots) = K \cdot 1/(1 - 2/3) = 3K$, at most about $3n$. Integer division nudges the ratio from $2/3$, changing the constant slightly but not the linear bound; with Java's growth from 10, the total stays below $3n$ throughout the first

three million adds.

Adding the n writes, the total is under $3n$ for doubling and about $4n$ for 1.5: constant per operation. □

A fixed increment reverses the picture. Starting at c and adding c when full, growth happens at c, $2c$, $3c$, ..., and the copies total

> $c + 2c + 3c + \ldots + (n/c)\cdot c \approx c \cdot (n/c)^2/2 = n^2/(2c)$

$O(n^2)$ in all, $O(n)$ per operation; a larger c only shrinks the constant. For 100 elements from capacity 10, adding 10 each time costs $10 + 20 + \ldots + 90 = 450$ copies, ×1.5 costs 202, doubling 150.

Goodrich and Tamassia explain amortization with the "accounting method": charge each cheap add a little more than it costs and bank the surplus as tokens. With doubling, the cheap adds between growths number half the array; at 2 tokens each, the bank can pay for copying every element at growth time. Since the balance never goes negative, each add costs a constant number of tokens.

Common mistake

Reading "amortized $O(1)$" as "every add is $O(1)$." A growing add still takes $O(n)$, a pause that can hurt a real-time system; if the size is known, preallocate: `new ArrayList<>(10000)`.

6.5.5 Using ArrayList, and what it costs

```
ArrayList<String> list =
        new ArrayList<>();
list.add("Ann");      // [Ann]
list.add("Dan");      // [Ann, Dan]
list.add(0, "Ben");   // [Ben, Ann, Dan]
list.get(1);          // "Ann"
list.remove(0);       // [Ann, Dan]
list.size();          // 2
```

Operation	Cost	Reason
`get(i)`, `set(i, e)`	$O(1)$	direct access by address calculation
`add(e)` (at the end)	amortized $O(1)$	growth is rare
`add(i, e)`, `remove(i)`	$O(n)$	later elements shift
`contains`, `indexOf`	$O(n)$	linear search
`size()`	$O(1)$	reads a counter field

Table 6.22: The cost of `ArrayList` operations.

The slides run this code and count the shifts:

```
ArrayList<String> list = new ArrayList<>();
list.add("Ann");
list.add("Dan");
list.add(0, "Ben");
String s = list.get(1);
list.remove(0);
int n = list.size();
```

Line	List	Capacity	Shifts	Cost
`new ArrayList<>()`	[ ]	0	—	*O(1)*
`add("Ann")`	[Ann]	10	0	first array allocated
`add("Dan")`	[Ann, Dan]	10	0	amortized *O(1)*
`add(0, "Ben")`	[Ben, Ann, Dan]	10	2 (right)	*O(n)*
`get(1)`	[Ben, Ann, Dan]	10	0	"Ann", *O(1)*
`remove(0)`	[Ann, Dan]	10	2 (left)	*O(n)*
`size()`	[Ann, Dan]	10	0	2, *O(1)*

Table 6.23: Trace of `ArrayList` usage.

All 4 shifts come from the two front operations, inserting and removing at index 0; in an array-based list every change at the front shifts the whole list, while appending, indexed reads, and size queries shift nothing. After `remove(0)` the capacity is still 10.

Against a plain array, `ArrayList` grows by itself and comes with methods for adding, removing, searching, and sorting; but it holds only objects, so primitives are boxed, and each access costs a call and a bounds check. For large fixed-size primitive data, an array fits better.

6.5.6 Code trace: add(i, e)

Inserting in the middle means opening a slot:

```
public void add(int i, E e) {
  Objects.checkIndex(i, size + 1);
  if (size == data.length) resize();
  for (int k = size; k > i; k--)
    data[k] = data[k - 1];
  data[i] = e;
  size++;
}
```

`Objects.checkIndex(i, size + 1)` checks $0 \le i \le size$ (the bound is `size + 1` so that appending is allowed) and throws `IndexOutOfBoundsException` otherwise. A full array is grown first. The loop runs backward from `k = size` to `k = i + 1`, copying each element one place right; then `data[i]`

receives the new element and the size goes up.

The loop must run from the *right*. Run from the left, `data[i + 1] = data[i]` would destroy the element in `data[i + 1]` at the first step; from the right, each element moves before anything overwrites it.

Step	Check / assignment	`data` (0–5)	`size`
start	—	[A, B, C, D, –, –]	4
index check	*0 ≤ 1 ≤ 4* ✓	[A, B, C, D, –, –]	4
full?	*4 < 6*: no growth	[A, B, C, D, –, –]	4
`k = 4`	`data[4] = data[3]`	[A, B, C, D, D, –]	4
`k = 3`	`data[3] = data[2]`	[A, B, C, C, D, –]	4
`k = 2`	`data[2] = data[1]`	[A, B, B, C, D, –]	4
`k = 1`	*1 > 1* fails: loop ends	[A, B, B, C, D, –]	4
`data[1] = e`	X written	[A, X, B, C, D, –]	4
`size++`	—	[A, X, B, C, D, –]	5

Table 6.24: Trace of `add(1, "X")` on `[A, B, C, D]` (capacity 6).

Cell 1 ends the loop holding a stale B, which the next line overwrites with X. There were *n − i = 4 − 1 = 3* shifts: all *n* elements at the front (*i = 0*), none at the end (*i = n*), *O(n)* in the worst case.

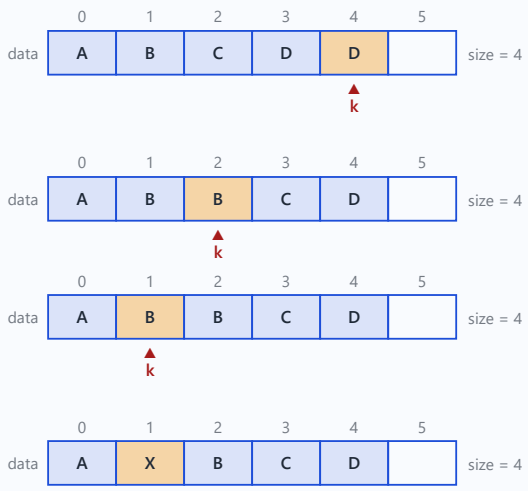


Figure 6.20: (a) `k = 4`: D copied right. (b) `k = 2`: B copied right. (c) Loop over; cell 1 holds a stale B. (d) X written: `[A, X, B, C, D]`.

6.5.7 Code trace: remove(i)

```
public E remove(int i) {
  Objects.checkIndex(i, size);
  E old = data[i];
  for (int k = i; k < size - 1; k++)
    data[k] = data[k + 1];
  data[--size] = null;
  return old;
}
```

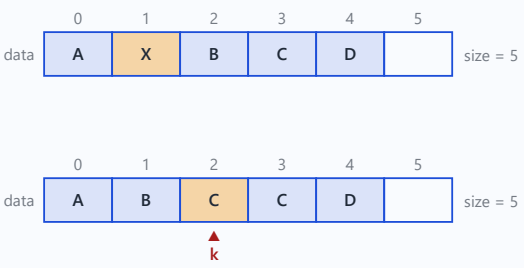
Removal reverses insertion. The index must satisfy $0 \le i < size$, since an existing element goes. It is saved in `old`, and the loop runs *left to right* from `k = i`, copying each element one place left; here starting at the left is right, because `data[i]` is meant to be overwritten.

Step	Check / assignment	`data` (0–5)	`size`
index check	$0 \le 1 < 5$ ✓	[A, X, B, C, D, –]	5
`old = data[1]`	`old = X`	[A, X, B, C, D, –]	5
`k = 1`	`data[1] = data[2]`	[A, B, B, C, D, –]	5
`k = 2`	`data[2] = data[3]`	[A, B, C, C, D, –]	5
`k = 3`	`data[3] = data[4]`	[A, B, C, D, D, –]	5
`k = 4`	$4 < 4$ fails: loop ends	[A, B, C, D, D, –]	5
`data[--size] = null`	size 4, `data[4] = null`	[A, B, C, D, –, –]	4
`return old`	returns X	[A, B, C, D, –, –]	4

Table 6.25: Trace of `remove(1)` on `[A, X, B, C, D]`.

After the loop, D sits in the last two cells. `data[--size] = null` lowers the size to 4 and clears cell 4. It may look pointless, as cell 4 is outside the list, but a leftover reference would keep D from being garbage-collected even after the list really dropped it. Such a "ghost" reference is a **memory leak**; the `null` prevents it.

There are $n - i - 1 = 5 - 1 - 1 = 3$ shifts: $n - 1$ when removing from the front, none (and $O(1)$) from the end ($i = size - 1$). The capacity is unchanged.



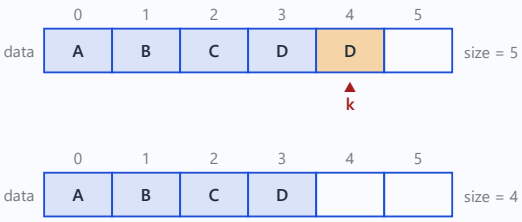


Figure 6.21: (a) `old = X`. (b) `k = 2`: C shifted left. (c) Loop over; D in the last two cells. (d) `data[4] = null`, size 4.

Common mistake

Running the shift loop from the left when inserting or from the right when removing. Each step then overwrites the next element, and part of the list fills with copies of one element. Rule: elements moving right, start right; moving left, start left. Java itself uses `System.arraycopy`, which is correct even when source and destination overlap.

6.6 LinkedList and the positional list

An array-based list is weak at the front, where every insertion or removal shifts all elements. A linked list avoids this by not storing elements contiguously. This section covers Java's `LinkedList`, the code of two of its operations, and the positional list ADT, which exploits linking fully.

6.6.1 The doubly linked list

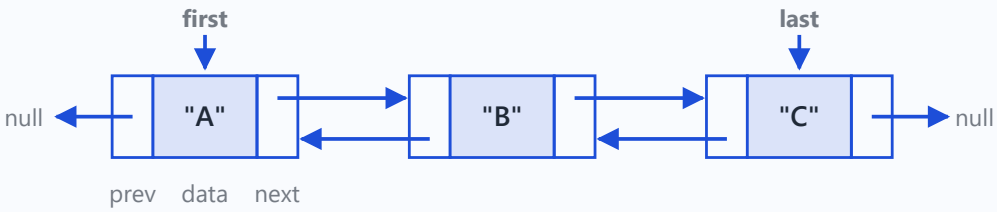


Figure 6.22: A three-element `LinkedList`. Each node holds a reference to the previous node, the data, and a reference to the next. The list keeps the first node in `first` and the last in `last`; the references at the ends are `null`.

In `LinkedList` (Figure 6.22) each element lives in its own **node**, holding the data and references to the previous (`prev`) and next (`next`) nodes; with links both ways, this is a **doubly linked list**. The list object knows only `first`, `last`, and the count `size`.

Two consequences follow. The ends are cheap: inserting at the front creates a node and points `first` at it, shifting nothing, and `last` makes appending walk-free too. Random access is expensive: with no address calculation for element 500, the nodes are walked link by link, from the nearer end but still in *O(n)*.

`LinkedList` also implements `Deque` (double-ended queue), so it has `addFirst`, `addLast`, `removeFirst`, `removeLast`, `push`, `pop`, `offer`, and `poll`, and works as a stack or a queue.

6.6.2 Using LinkedList, and what it costs

```
LinkedList<String> list =
        new LinkedList<>();
list.add("B");          // [B]
list.addFirst("A");     // [A, B]
list.addLast("C");      // [A, B, C]
list.get(1);            // "B" (by walking)
list.removeFirst();     // [B, C]
list.size();            // 2
```

Operation	Cost
add at front / end, remove from front / end	*O(1)*
`get(i)`, `set(i, e)`	*O(n)*
`add(i, e)`, `remove(i)`	*O(n)**
add / remove through an iterator	*O(1)*

Table 6.26: The cost of `LinkedList` operations.

The asterisk matters. Linking a node in or out is a few assignments, *O(1)*, but `add(i, e)` and `remove(i)` must first *find* node i in *O(n)*. "Linked-list insertion is fast" holds only when the position is already at hand: at the ends, or at a `ListIterator`.

The slides trace this code:

```
LinkedList<String> list = new LinkedList<>();
list.add("B");
list.addFirst("A");
list.addLast("C");
String s = list.get(1);
list.removeFirst();
int n = list.size();
```

Line	List	`first` / `last`	Work	Cost
`new LinkedList<>()`	[]	`null` / `null`	—	*O(1)*

Line	List	`first` / `last`	Work	Cost
`add("B")`	[B]	B / B	`add` = `addLast`	*O(1)*
`addFirst("A")`	[A, B]	A / B	`A.next = B`, `B.prev = A`	*O(1)*
`addLast("C")`	[A, B, C]	A / C	no walk, thanks to `last`	*O(1)*
`get(1)`	[A, B, C]	A / C	*1 < 3/2 = 1* fails: 1 back from `last`	*O(n)*
`removeFirst()`	[B, C]	B / C	`B.prev = null`	*O(1)*
`size()`	[B, C]	B / C	counter field	*O(1)*

Table 6.27: Trace of `LinkedList` usage.

At `get(1)`, `size >> 1` (the integer part of *3/2*) is 1; *1 < 1* is false, so the walk starts at `x = last` (C) and one `x = x.prev` reaches B. `removeFirst()` changes two references: `first` to B, B's `prev` to `null`. A is now unreferenced and garbage-collected.

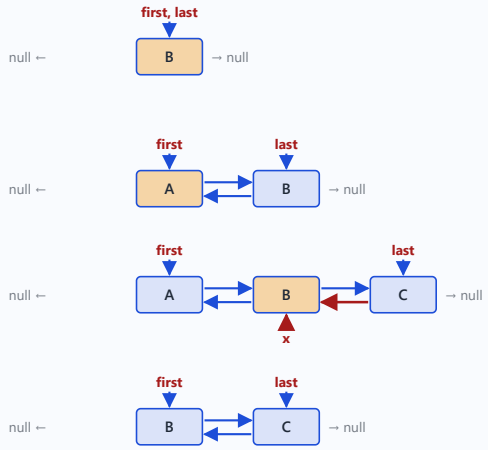


Figure 6.23: (a) `add("B")`: one node, both first and last. (b) `addFirst("A")`: new node linked at the front. (c) `get(1)`: one link back from `last`. (d) `[B, C]` after `removeFirst()`.

6.6.3 Code trace: addLast(e)

Appending, in the manner of Java's `linkLast`:

```
void addLast(E e) {
  Node<E> l = last;
  Node<E> n = new Node<>(l, e, null);
  last = n;
  if (l == null) first = n;
  else l.next = n;
  size++;
}
```

Five steps: save the old last node in `l`; create the node with the previous node (`l`), the data (`e`), and no next (`null`), so the backward link exists from the start; point `last` at it; if the list was empty (`l == null`) point `first` at it too, otherwise attach it to the old last node's forward link; increment `size`.

Line	1st call ("A")	2nd call ("B")
`l = last`	`l = null`	`l` = node(A)
`n = new Node<>(l, e, null)`	n: prev `null`, next `null`	n: prev node(A), next `null`
`last = n`	`last` = node(A)	`last` = node(B)
`if (l == null)`	true: `first` = node(A)	false: `A.next` = node(B)
`size++`	1	2

Table 6.28: `addLast("A")` and `addLast("B")` on an empty list.

The result is `first → A ⇄ B ← last`. Nothing was traversed, so appending is *O(1)* whatever the length, thanks to `last`; with only `first`, finding the end would make it *O(n)*.

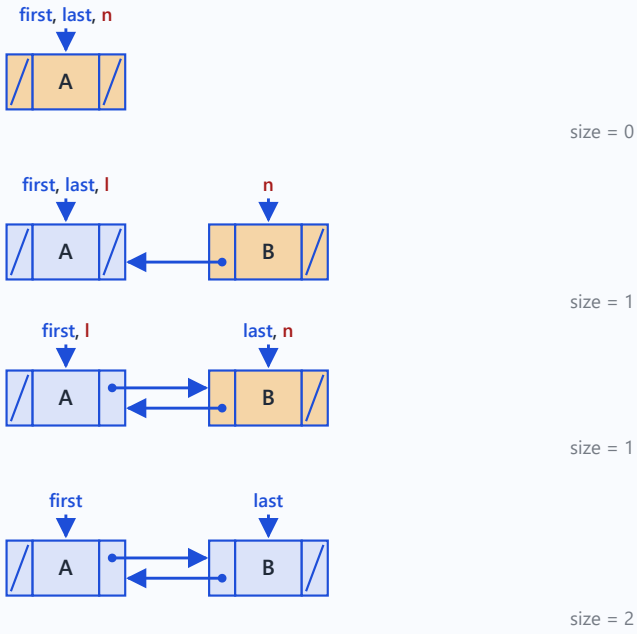


Figure 6.24: (a) First call: one node, both `first` and `last`. (b) Second call: the new node's backward link points to A. (c) `l.next = n`: forward link from A to B. (d) A two-node list, `size = 2`.

Common mistake

Forgetting the empty list. Without `if (l == null) first = n;`, the first add runs `l.next = n` on a `null` reference and throws `NullPointerException`. Always test linked-list code on an empty list, a one-element list, and the end nodes.

6.6.4 Code trace: remove(i)

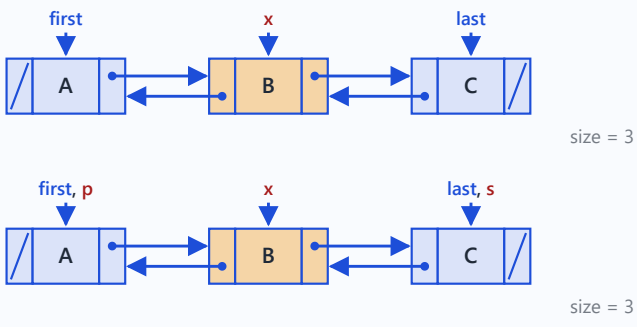
```
E remove(int i) {
  Objects.checkIndex(i, size);
  Node<E> x = first;
  for (int k = 0; k < i; k++) x = x.next;
  Node<E> p = x.prev, s = x.next;
  if (p == null) first = s; else p.next = s;
  if (s == null) last = p; else s.prev = p;
  size--;
  return x.data;
}
```

Removal by index has two phases. First `x` starts at the first node and follows `next` `i` times. Then the node is unlinked: with its neighbors saved as `p` and `s`, the predecessor's forward link goes to `s` and the successor's backward link to `p`, bypassing `x` both ways. If `x` is first (`p == null`), `first` is updated; if last (`s == null`), `last`.

Line	State	Note
`checkIndex(1, 3)`	*0 ≤ 1 < 3* ✓	—
`x = first`	`x` = A	no direct access as in an array
`k = 0`	`x = x.next` → B	1 link walked
`k = 1`	*1 < 1* fails	walk ends: *O(i)*
`p = x.prev`, `s = x.next`	`p` = A, `s` = C	the two neighbors
`p.next = s`	A → C	`p` not `null`
`s.prev = p`	C → A	`s` not `null`; B is unlinked
`size--`	`size = 2`	—
`return x.data`	returns "B"	list is `[A, C]`

Table 6.29: Trace of `remove(1)` on `[A, B, C]`.

The four linking lines are *O(1)*, the walk before them *O(i)*, *O(n)* at worst. The slides' code always walks from the front; Java's walks back from `last` when *i ≥ size/2*, capping the walk at *n/2*, which still leaves the growth rate unchanged.



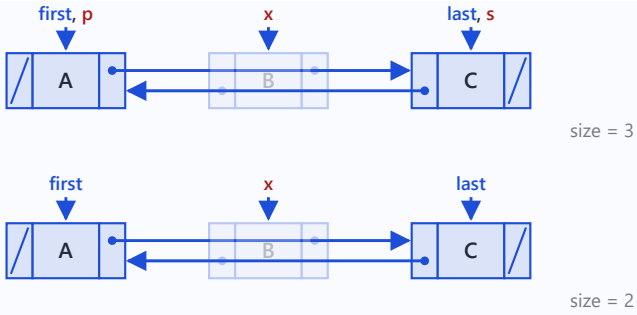


Figure 6.25: (a) `x` reached B after one link. (b) Neighbors: `p` = A, `s` = C. (c) Both links bypass B, now faded. (d) `size = 2`; B returned.

6.6.5 The positional list ADT

Indices suit arrays, not linked lists, and they are unstable: one insertion at the front shifts every index, so "element 5" then names a different element. A text editor's cursor stays beside its word when text is inserted before it; it is a *place*, not a number. Goodrich, Tamassia, and Goldwasser make this idea an ADT.

Definition 6.5

A **position** is an object representing an element's place in a list, valid and unaffected by other insertions and removals until that element is removed. A **positional list** is a list ADT accessed through positions rather than indices, inserting, removing, and replacing at a given position.

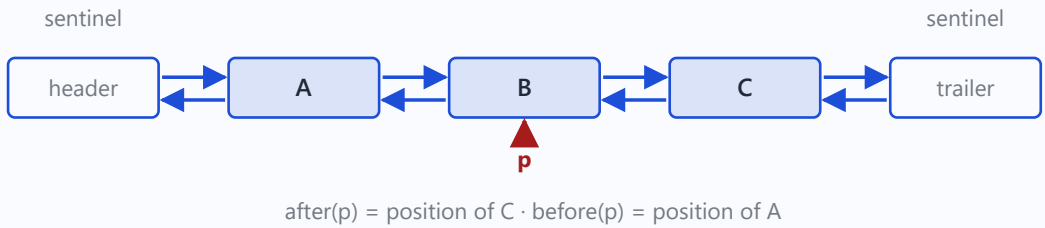


Figure 6.26: A positional list with sentinels: `header` and `trailer` hold no data. `p` is B's position; `after(p)` gives C's, `before(p)` A's.

In the book, `Position<E>` has one method, `getElement()`. In the linked implementation the node itself is the position, its class implementing `Position`; users can read the element and pass the position back, but not see the node's fields.

The second key idea is **sentinel** nodes, `header` and `trailer`, which hold no data and exist even in an empty list, pointing to each other. Every real node then has neighbors on both sides, insertion is always between two nodes, and `null` checks ("empty? first? last?") vanish.

Method	Cost
`first()`, `last()`, `before(p)`, `after(p)`	*O(1)*
`addFirst(e)`, `addLast(e)`	*O(1)*
`addBefore(p, e)`, `addAfter(p, e)`	*O(1)*
`set(p, e)`, `remove(p)`	*O(1)*

Table 6.30: Positional list methods and their costs.

Every method is constant-time because the position is the node: nothing is searched, only neighboring links rewired. The standard library has no direct counterpart. `ListIterator` comes closest, with *O(1)* insertion and removal at its cursor, but a cursor stands in one place, and two iterators making structural changes cannot share a list.

6.6.6 Code trace: addAfter(p, e)

```
Position<E> addAfter(Position<E> p, E e) {
  Node<E> n = validate(p);
  return addBetween(e, n, n.next);
}
Position<E> addBetween(E e, Node<E> pred, Node<E> succ) {
  Node<E> newest = new Node<>(pred, e, succ);
  pred.next = newest;
  succ.prev = newest;
  size++;
  return newest;
}
```

`addAfter` uses `validate(p)` to check that the position is a valid node of this list and convert it to the node type, then calls `addBetween` to insert between `n` and `n.next`. `addBetween` creates the node already pointing at both neighbors, points the neighbors at it, increments the size, and returns the node: the new element's position.

Line	State
`validate(p)`	`n` = node(A)
`addBetween(e, n, n.next)`	`pred` = A, `succ` = C
`newest = new Node<>(pred, e, succ)`	B's prev is A and next is C, but nothing points to B yet
`pred.next = newest`	forward: A → B → C
`succ.prev = newest`	backward: C → B → A; the old A ⇄ C links are gone
`size++`	`size = 3`
`return newest`	returns B's position

Table 6.31: `addAfter(p, "B")` on `header ⇄ A ⇄ C ⇄ trailer` (`p` is the position of A).

Four assignments, no traversal, no shifting: *O(1)*. Order matters: the new node points to its neighbors before their links change. Reversed — `pred.next = newest` first, then `newest.next = pred.next` — the node would point to itself.

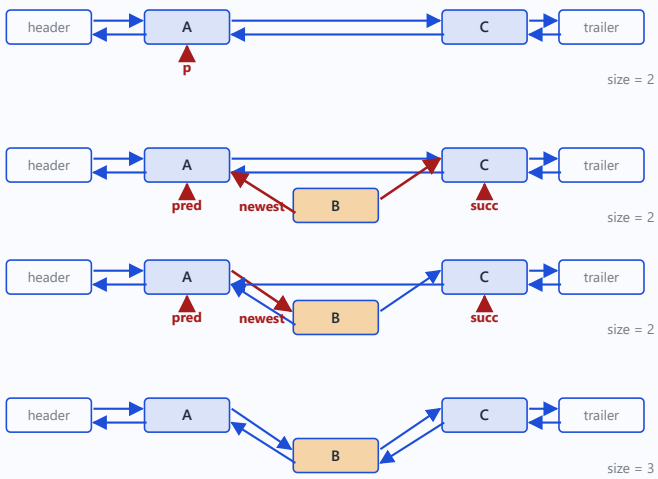


Figure 6.27: (a) Start: `p` is A's position. (b) The new node points to both neighbors. (c) `pred.next = newest`. (d) `succ.prev = newest`, `size = 3`: B linked both ways.

`remove(p)` is *O(1)* for the same reason: the position is the node, whose neighbors are read from its links and joined. `LinkedList.remove(int)` must walk to the index first, *O(n)*. The difference lies in the argument, not the data structure.

Common mistake

Reusing a removed position. After `remove(p)`, a good implementation nulls the node's links so that `validate` throws; without that, `addAfter(p, e)` could corrupt the list through a node already cut out.

6.6.7 ArrayList or LinkedList?

	`ArrayList`	`LinkedList`
Internal structure	growable array	doubly linked nodes
`get(i)`	*O(1)*	*O(n)*
Append	amortized *O(1)*	*O(1)*
Insert / remove at front	*O(n)*	*O(1)*
Memory per element	1 reference (+ spare capacity)	a node object (3 references)
Cache	contiguous, fast	scattered, slow

Table 6.32: `ArrayList` and `LinkedList` compared.

On paper it looks balanced; in practice `ArrayList` wins. Most programs mostly read and append, where it is at least as good, and `LinkedList`'s advantages are often eaten by memory and cache effects (Section 6.6.8). `ArrayList` is the default.

For heavy use at both ends, as a queue or deque, `LinkedList` is an option, though `ArrayDeque`, a circular array with amortized *O(1)* at both ends and no per-node overhead, is usually better. `LinkedList` shines when a `ListIterator` sweep makes many insertions and removals.

6.6.8 Two lists in memory

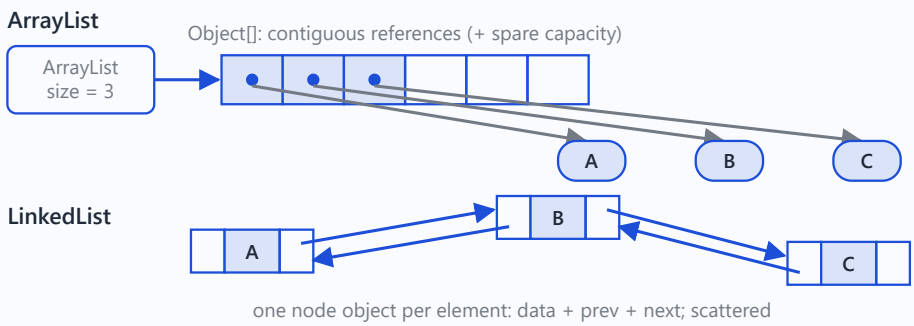


Figure 6.28: Memory layout. Top, `ArrayList`: a contiguous array of references to the element objects. Bottom, `LinkedList`: a separate node per element, scattered.

`ArrayList` spends about one reference (4 or 8 bytes, depending on the JVM) per element, plus spare capacity. `LinkedList` adds a node per element, with a header and three references, about 24–40 bytes: several times the memory.

The bigger difference is the **cache**. The processor fetches memory in blocks into its fast cache. `ArrayList`'s contiguous references arrive several at a time, so traversal runs from the cache; `LinkedList`'s scattered nodes mean each `next` may hit an address not in the cache, a **cache miss**, and main memory is tens of times slower. Both traverse in *O(n)*, but `ArrayList` is usually clearly faster in real time.

6.7 Other list classes

`ArrayList` and `LinkedList` cover almost everything. Four more structures serve special cases or survive from older versions; most concern lists shared by threads.

6.7.1 Vector

`Vector` is a growable array like `ArrayList`, except that it doubles its capacity by default and all its methods are `synchronized`.

`synchronized` guards a method with a **lock**: while one thread runs it, other threads wait to enter any synchronized method of the same object. So two threads cannot `add` at once and corrupt the array; single operations are thread-safe.

```
Vector<Integer> v = new Vector<>();
v.add(10);            // [10]
v.add(20);            // [10, 20]
v.get(0);             // 10
v.remove(0);          // [20]
v.size();             // 1
v.capacity();         // 10

// compound operation: needs an outside lock
synchronized (v) {
    if (!v.isEmpty()) v.remove(0);
}
```

`v.capacity()` returns 10: the first array has 10 cells and never shrinks. Every call pays for the lock, even with one thread. Worse, locks guard only single methods; a compound check-then-act such as "remove the first element if not empty" is still unsafe, hence the `synchronized (v) { … }` block at the end.

`Vector` dates from Java 1.0. New code uses `ArrayList` for one thread and `Collections.synchronizedList(...)` or `CopyOnWriteArrayList` for several.

6.7.2 The check-then-act race

One interleaving shows why the compound operation is unprotected. Let `v = [20]`, and let threads T1 and T2 both run `if (!v.isEmpty()) v.remove(0);`.

Step	T1	T2	v
1	`isEmpty()` → `false`	—	[20]
2	—	`isEmpty()` → `false`	[20]
3	`remove(0)` → 20	—	[ ]
4	—	`remove(0)` → `ArrayIndexOutOfBoundsException`	[ ]

Table 6.33: One interleaving of two threads. Each method is locked on its own, but the gap between two calls is unprotected.

T1 locks `isEmpty()`, sees `false`, and releases the lock; the scheduler may now run T2, which also sees one element. Both decide to remove: T1 succeeds, T2 throws on the empty vector. Each method is synchronized, but the pair is not **atomic**, not one indivisible unit. An error whose outcome depends on thread interleaving is a **race condition**. The fix puts the whole operation under one lock:

```
synchronized (v) { if (!v.isEmpty()) v.remove(0); }
```

The block takes the very lock `Vector`'s methods use. While T1 is inside, T2 waits; then it sees an empty vector and does not remove.



Figure 6.29: (a) T2 slipped in; both saw a non-empty vector. (b) T1 removed; T2's removal failed. (c) The fix: while T1 holds the lock, T2 cannot enter. (d) With the lock, T2 sees an empty vector and does not remove.

Common mistake

Thinking "the class is thread-safe, so my code is." `Vector`, `Collections.synchronizedList`, and the like protect single calls only; every multi-call operation (check and remove, add if absent, traverse and modify) needs its own lock.

6.7.3 The Stack class

`Stack` extends `Vector` with a **last-in, first-out** (LIFO) stack whose operations touch only the top, handy for undo/redo, where the last action is undone first.

Method	Purpose
`push(e)`	adds on top
`pop()`	removes and returns the top
`peek()`	returns the top without removing it
`empty()`	is the stack empty?

Method	Purpose
`search(o)`	distance from the top, counting from 1; −1 if absent

Table 6.34: `Stack` methods.

`pop` or `peek` on an empty stack throws `EmptyStackException`. `search` counts from 1 at the top: after pushing `a`, `b`, `c`, `search("c")` is 1 and `search("a")` is 3.

`Stack` has a design flaw: it inherits `Vector`'s `get(i)`, `add(i, e)`, and `remove(i)`, which reach into the middle and break LIFO (`add(0, "z")` slips an element in at the *bottom*), and every operation pays for synchronization. The documentation recommends `Deque` for stacks:

```
Deque<E> s = new ArrayDeque<>();
```

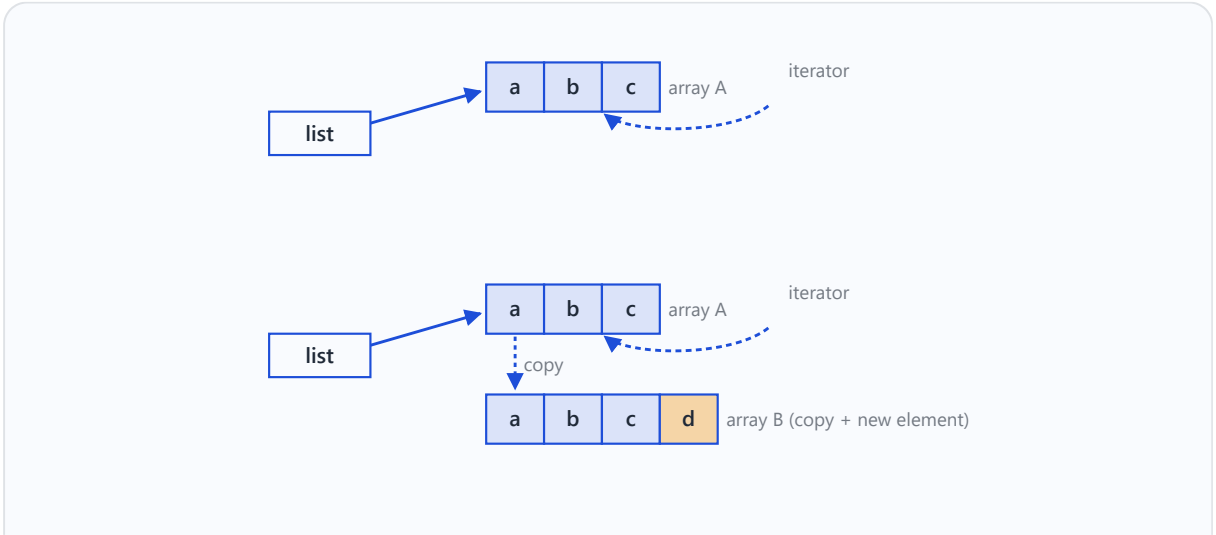
`ArrayDeque`'s `push`, `pop`, and `peek` mean the same, without locks or access to the middle.

6.7.4 CopyOnWriteArrayList

Some lists are read constantly but rarely changed, like the **event listeners** notified when a button is clicked: traversed on every event, seldom extended. With many threads traversing, waiting for locks is costly. `CopyOnWriteArrayList` is built for this.

The idea is in the name: a change (`add`, `set`, `remove`) copies the internal array, edits the copy, and repoints the list; the old array is never touched. Consequences:

- **Reads are lock-free and fast.** A reader reads the current array, which never changes in place.
- **Writes are expensive.** Each copies the whole array, *O(n)*; writers queue on a short lock so that none overwrites another's copy.
- **Iterators traverse a snapshot**, the array current when they were created. Later writes leave them on the old array, so `ConcurrentModificationException` never occurs; in exchange they miss new elements and `it.remove()` is unsupported.



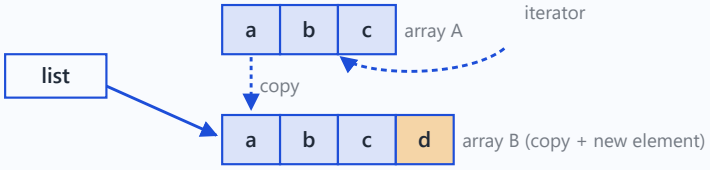


Figure 6.30: Copy on write. (a) The list and an iterator share array A. (b) `add("d")`: the writer copies A and adds to the copy, B. (c) The list now points to B; the old iterator still traverses A.

Good for lists traversed often, changed rarely, and shared by threads: listeners, configuration. Bad for large, frequently changed lists, where every add copies everything.

6.7.5 Code trace: copy on write

```
public boolean add(E e) {
  synchronized (lock) {
    Object[] old = array;
    int n = old.length;
    Object[] copy = Arrays.copyOf(old, n + 1);
    copy[n] = e;
    array = copy;      // a single reference assignment
    return true;
  }
}
```

`synchronized (lock)` serializes writers; readers never touch it. `old = array` grabs the current array; `Arrays.copyOf(old, n + 1)` copies it into one cell more; `copy[n] = e` changes only the copy. `array = copy` is a single reference assignment, and since that is indivisible in Java, a reader sees the old array or the new one, never a half-built one.

Step	List points to	Iterator	Note
start	A = [a, b, c]	A, `cursor = 1`	"a" has been read
lock taken	A	A	readers do not wait
`old = array`, `n = 3`	A	A	—
`copyOf(old, 4)`	A	A	B = [a, b, c, –]; *O(n)*
`copy[3] = "d"`	A	A	B = [a, b, c, d]
`array = copy`	B	A	a single reference assignment
`return true`	B	A	lock released
`it.next()` ×2	B	A: "b", "c"	"d" not seen; no exception
new `iterator()`	B	B	traverses [a, b, c, d]

Table 6.35: `add("d")` on `[a, b, c]` while an iterator has read "a".

When the old iterator finishes, array A is unreferenced and garbage-collected. A for-each loop behaves alike: add "e" to `[a, b, c, d]` on seeing "a", and the loop still prints a, b, c, d without an exception; afterwards the list is `[a, b, c, d, e]`.

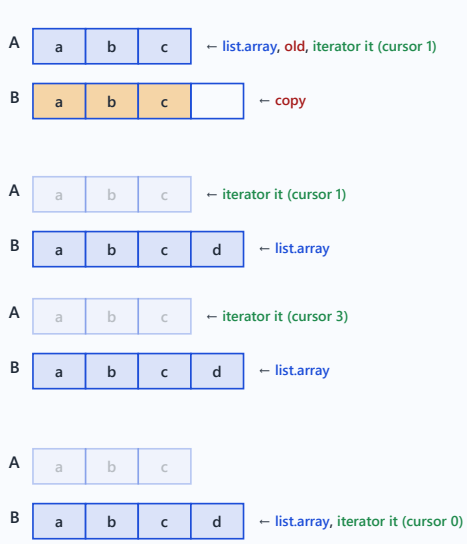


Figure 6.31: (a) `Arrays.copyOf`: B created, A untouched. (b) `array = copy`: the list points to B, the iterator still to A. (c) The iterator reached A's end. (d) A new iterator traverses B.

6.7.6 Arrays.asList: a list view of an array

To pass an array where a `List` is expected, `Arrays.asList(array)` returns, without copying, a list view sharing the array's memory.

```
String[] array = {"a", "b", "c"};
List<String> list = Arrays.asList(array);
list.set(0, "x");      // works: array[0] is now "x" too
array[2] = "z";         // the list changes too → [x, b, z]
list.add("d");          // UnsupportedOperationException (fixed size)
List<String> copy = new ArrayList<>(Arrays.asList(array));   // independent copy
```

Two rules. It is **fixed-size**: the array cannot grow, so `add` and `remove` throw `UnsupportedOperationException`. It is **not immutable**: `set` just writes an array cell, and changes flow both ways.

	`set`	`add` / `remove`	**Tie to the array**
`Arrays.asList(array)`	✓	✗	shared
`List.of(...)`	✗	✗	none (immutable)
`new ArrayList<>(...)`	✓	✓	none (copy)

Table 6.36: Lists created three ways, compared.

The slides also show how to get an independent, growable copy:

```
String[] array = {"a", "b", "c"};
List<String> list = Arrays.asList(array);
list.set(0, "x");
array[2] = "z";
// list.add("d");   → error (fixed size)
List<String> copy = new ArrayList<>(list);
copy.add("d");
System.out.println(list + " " + copy);
```

Line	`array`	`list`	`copy`
`String[] array = …`	[a, b, c]	—	—
`Arrays.asList(array)`	[a, b, c]	[a, b, c] (same array)	—
`list.set(0, "x")`	[x, b, c]	[x, b, c]	—
`array[2] = "z"`	[x, b, z]	[x, b, z]	—
`new ArrayList<>(list)`	[x, b, z]	[x, b, z]	[x, b, z] (separate array, capacity 3)
`copy.add("d")`	[x, b, z]	[x, b, z]	[x, b, z, d] (capacity 4)

Table 6.37: Trace of `Arrays.asList`.

It prints `[x, b, z] [x, b, z, d]`. `Arrays.asList` is *O(1)*; `new ArrayList<>(list)` copies into an array exactly as long as the list (*O(n)*), which grows to *3 + 3/2 = 4* cells on the first add. The copy changes freely without touching the array or the view.

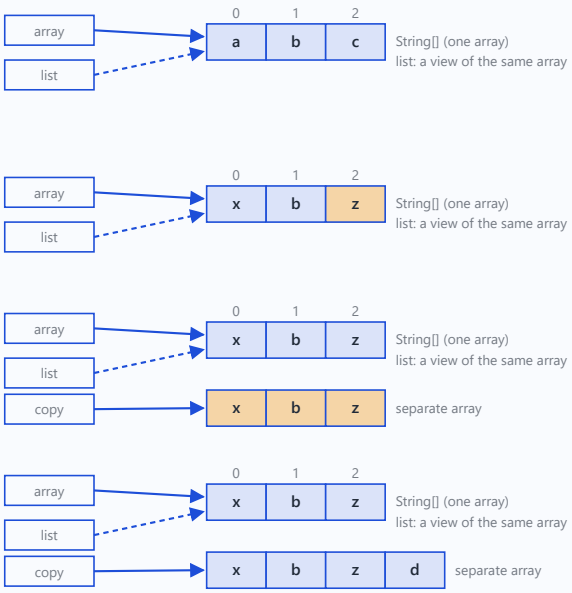


Figure 6.32: (a) `Arrays.asList` : `list` views the same array. (b) `array[2] = "z"` shows through the list. (c) `new ArrayList<>(list)` allocates a separate array. (d) "d" added to the copy; array and view unchanged.

Common mistake

Passing a primitive array to `Arrays.asList`. `int[] d = {1, 2, 3}; Arrays.asList(d)` gives not a three-element `List<Integer>` but a one-element `List<int[]>`, with `size()` 1: a type parameter cannot be primitive, so the array itself is the element. Use `Integer[]` or `Arrays.stream(d).boxed().toList()`.

6.8 Summary

- A **list** keeps elements in positional order, indexed from 0, duplicates allowed. Basic operations: append, insert, remove, index, slice, search, length. Length (`size`) is not capacity.
- In Java an ADT is an interface, an implementation a class. `Iterable` → `Collection` → `List`: for-each, common operations without indices, then position-based methods. Declare variables by interface: `List<String> l = new ArrayList<>();`.
- `subList` is a **view**: changes reach the original, and outside structural changes invalidate it. `List.of` / `List.copyOf` are **immutable**, `Collections.unmodifiableList` only **read-only**, `new ArrayList<>(…)` an independent copy.
- `remove(int)` removes by index, returning the element; `remove(Object)` by value, returning a `boolean`; in a `List<Integer>` a primitive argument is an index. Type parameters cannot be primitive; boxing is automatic, and unboxing `null` throws `NullPointerException`.
- Traverse with an indexed loop, for-each/forEach, or an iterator; on a `LinkedList` the first is $O(n^2)$, the others $O(n)$. An iterator is a cursor between elements.
- Outside modification during traversal trips the fail-fast `modCount` check: `ConcurrentModificationException`. Remove safely with `it.remove()` (once per `next()`) or `removeIf`. `ListIterator` moves both ways and can `add`, `set`, `remove`; `Spliterator` splits for parallel work.
- `ArrayList`: a growable array; `get` / `set` $O(1)$, append amortized $O(1)$, insert/remove $O(n)$; grows about 1.5 times when full (10 on the first add). Geometric growth costs $O(n)$ copies in all, a fixed increment $O(n^2)$.
- `LinkedList`: doubly linked; $O(1)$ at the ends, $O(n)$ by index. Relinking is cheap, finding the place expensive. The **positional list**, with the node as position and sentinel nodes, makes every operation $O(1)$.
- `ArrayList` is the default; memory and cache behavior favor it. For heavy use at both ends, consider `ArrayDeque` or `LinkedList`.
- `Vector` and `Stack` are legacy synchronized classes that leave compound operations unprotected; use `ArrayDeque` for stacks. `CopyOnWriteArrayList` copies on write and iterates over snapshots. `Arrays.asList` is a fixed-size view of the array: `set` works, `add` / `remove` do not.

Class	`get(i)`	Append	Front / middle	Threads
`ArrayList`	O(1)	amortized O(1)	O(n)	not safe
`LinkedList`	O(n)	O(1)	O(1) / O(n)	not safe
`Vector`	O(1)	amortized O(1)	O(n)	synchronized
`Stack`	O(1)	`push` O(1)	—	synchronized
`CopyOnWriteArrayList`	O(1)	O(n)	O(n)	safe (by copying)
`Arrays.asList`	O(1)	✗	✗	not safe

Table 6.38: The list classes at a glance.

6.9 Key terms

Each term links to the course glossary, which gives its Turkish equivalent and its use in other chapters.

list · collection · abstract data type (ADT) · interface · generics · index · capacity · view · mutable · immutable · persistent data structure · autoboxing · unboxing · traversal · iterator · cursor · amortized analysis · linked list · doubly linked list · node · position · positional list · sentinel · null · deque

6.10 Test Yourself

1. Let `l = [A, B, C, D, E, F]` be an `ArrayList`. We run `l.subList(1, 4).clear();` and then `l.subList(0, 2).set(1, "Q");`. What is `l` at the end?

   Answer

   The first view, *[1, 4)*, is B, C, D, and `clear()` removes them: `[A, E, F]`. The second view, *[0, 2)*, is `[A, E]`; `set(1, "Q")` changes its element 1, index *0 + 1 = 1* of the original. Result: `[A, Q, F]`.

2. On `List<Integer> m = new ArrayList<>(List.of(4, 0, 2, 1));` we call `m.remove(m.get(1));` and then `m.remove(1);`. What is the list?

   Answer

   `m.get(1)` returns an `Integer` (0), so `remove(Object)` removes the *value 0*: `[4, 2, 1]`. `m.remove(1)` passes a primitive `int`, so `remove(int)` removes 2 at index 1: `[4, 1]`.

3. Twenty-five elements are added to an empty list created with `new ArrayList<>()`. At which adds

does the capacity grow, and to what? How many elements are copied in total?

Answer

0 to 10 at the 1st add (no copies), 10 to 15 at the 11th (10 copies), 15 to 22 at the 16th (15), 22 to 33 at the 23rd (22): *10 + 15 + 22 = 47* copies, ending at size 25, capacity 33.

4. How many links are walked in total if a 7-element `LinkedList` is traversed with an indexed `for` loop using `get(i)`? How many with an iterator?

Answer

The integer part of *n/2* is 3. *i = 0, 1, 2* walk 0, 1, 2 links from the front; *i = 3, 4, 5, 6* walk *7 − 1 − i* = 3, 2, 1, 0 from the end: *0 + 1 + 2 + 3 + 2 + 1 + 0 = 9*. An iterator walks one link per `next()`: *n − 1 = 6*.

5. On `r = [P, Q, R]` we take `ListIterator<String> it = r.listIterator(3);` and call, in order, `it.previous()`, `it.previous()`, `it.set("X")`, `it.add("Y")`, `it.next()`. What does the list become, and what does the last call return?

Answer

Start: `P Q R |`. The two `previous()` calls return R, then Q: `P | Q R`. `set("X")` replaces the last returned Q: `P | X R`. `add("Y")` inserts before the cursor: `P Y | X R`. `next()` returns X. The list is `[P, Y, X, R]`.

6. We take a five-element `ArrayList`'s spliterator as `a` and call `Spliterator b = a.trySplit();`. Which indices do `a` and `b` cover?

Answer

The fence becomes 5 and the midpoint *(0 + 5) / 2 = 2*. The first half goes to the new spliterator: `b` covers *[0, 2)* (indices 0, 1), `a` *[2, 5)* (2, 3, 4).

7. A hundred elements are added to an empty dynamic array of capacity 10. How many copies are made if the capacity, whenever it fills, (a) grows by 10, (b) becomes *old + old/2*?

Answer

(a) Growth at 10, 20, …, 90 elements: *10 + 20 + … + 90 = 450* copies. (b) 10 → 15 → 22 → 33 → 49 → 73 → 109: *10 + 15 + 22 + 33 + 49 + 73 = 202*. A fixed increment grows quadratically, proportional growth linearly.

8. Let `Stack<String> s` be an empty stack. After `s.push("a"); s.push("b"); s.push("c"); s.add(0, "z");`, what does `s.pop()` return, and what is `s.search("z")` afterwards? What design problem does this show?

   Answer

   `add(0, "z")`, inherited from `Vector`, puts "z" at the bottom: `[z, a, b, c]`. `pop()` returns "c", leaving `[z, a, b]`; `search("z")` counts b 1, a 2, z 3. Inheriting from `Vector` gives `Stack` methods that break LIFO.

9. A `CopyOnWriteArrayList` `c = [a, b, c, d]` is traversed with for-each; the body calls `c.add("e")` if the element is "a" and prints every element. What is printed, and is there an exception? What would happen with an `ArrayList`?

   Answer

   The iterator traverses its snapshot. Adding "e" repoints the list at a new array while the iterator continues on the old one: `abcd` is printed, no exception, and the list ends as `[a, b, c, d, e]`. With an `ArrayList`, the add would bump `modCount` and the next `next()` would throw `ConcurrentModificationException`.

10. For `int[] d = {1, 2, 3};`, what does `Arrays.asList(d).size()` return, and why?

    Answer

    1. A type parameter cannot be primitive, so the `int`s are not boxed; the array itself is the single element of a `List<int[]>`. For three elements, declare `Integer[]`.

6.11 Further reading

- M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, *Data Structures and Algorithms in Java* (6th ed., 2014), Chapter 7: the list ADT, array lists and their amortized analysis, positional lists, and iterators, in Java close to this chapter's `addBetween` code.
- J. Bloch, *Effective Java* (3rd ed., 2018): items on referring to objects by interface, preferring for-each, preferring primitives to boxed types, and minimizing mutability explain this chapter's design advice, from a designer of the Collections Framework.
- Oracle, Java SE 17 API documentation: the pages for `List`, `ArrayList`, `LinkedList`, `ListIterator`, and `CopyOnWriteArrayList` state each method's contract, its exceptions, and the limits of fail-fast iterators precisely.

6.12 References

1. J. Bloch, *Effective Java*, 3rd ed., Addison-Wesley, 2018.
2. M. T. Goodrich, R. Tamassia, M. H. Goldwasser, *Data Structures and Algorithms in Java*, 6th ed., Wiley, 2014.
3. Oracle, *Java SE 17 API Specification*: documentation for `java.util.Collection`, `List`, `Iterator`, `ListIterator`, and `Spliterator`.

PART II · CHAPTER 7

Stacks

The stack is among the simplest data structures and among the most useful. Items enter and leave at one end only, so the last one in is the first one out. What looks like a restriction is what lets every basic operation run in constant time, and it suits the many problems of the form "close the most recently opened thing first." We start from the abstract definition, implement the stack with an array and with a linked list, and see why a dynamic array grows by doubling. Then come applications: browser history, the call stack, balanced parentheses, next greater element, Towers of Hanoi, and stock span. The chapter ends by converting arithmetic expressions between infix, prefix, and postfix notation and evaluating postfix expressions.

Code for this chapter: `github.com/sercankulcu/data-structures-java/Bolum07/src`

LEARNING OUTCOMES

The LIFO model
Explain last-in, first-out behavior and give a stack's state and return values after a sequence of operations.

Implementation
Implement a stack with an array and a linked list, handling overflow, underflow, and dynamic growth correctly.

Applications
Use a stack for balanced parentheses, expression conversion and evaluation, call management, back/forward navigation, next greater element, and stock span.

Cost analysis
Give and justify the time and space cost of the basic operations; compute the amortized cost of doubling.

7.1 The stack ADT

A data structure can be described at two levels. The first says *what* it does: which operations exist, what each changes and returns. The second says *how*: how the data is laid out in memory and what steps each operation takes. A first-level description is an **abstract data type** (ADT). This section treats the stack as an ADT; implementation comes next.

7.1.1 What is a stack?

Picture a pile of clean plates. A washed plate always goes on top, and you take plates from the top. To reach the bottom one you must lift off every plate above it. Nobody pulls a plate from the middle; it is awkward, and the pile falls over.

A **stack** imitates this arrangement. Items are kept in sequence, but only one end, the **top**, is accessible, and both insertion and removal happen there. The item that comes off is therefore always the most recently added one still on the stack.

Definition 7.1

A **stack** is an abstract data type that keeps items in sequence and allows insertion and removal only at one end, the top. This discipline is called **LIFO** (last-in, first-out), or equivalently **FILO** (first-in, last-out).

Push 1, 2, 3 and then pop three times, and you get 3, 2, 1: the last in came out first (LIFO), the first in came out last (FILO).

The discipline turns up all over software:

- **Undo**: A text editor pushes every change onto a stack. “Undo” reverses the latest change, then the one before, so changes are undone in reverse order.
- **The browser’s “back” button**: Visited pages sit on a stack, and “back” returns to the page you left last. Section 7.3.2 develops this with two stacks.
- **Function calls**: If `main` calls `f` and `f` calls `g`, then `g` finishes first, then `f`, then `main`. Languages therefore track calls on a stack (7.3.3).

Each case is about the most recently started job that has not finished. When it ends, we return to the job started just before it, and a stack keeps this list of waiting jobs as cheaply as possible.

Common mistake

Thinking of a stack as a list you can read in any order. Only the top item is visible; reaching a middle item means popping everything above it. The stack ADT has no “read the third item”

operation.

7.1.2 Basic operations

Five operations define the stack ADT. Their names are standard and match the Java library: `push` adds an item on top, `pop` removes the top item.

Operation	Description
`push(x)`	Adds a new item `x` on top.
`pop()`	Removes and returns the top item.
`top()`	Returns the top item without removing it (`peek()` in the Java library).
`isEmpty()`	Returns `true` if the stack is empty, else `false` .
`size()`	Returns the number of items.

Table 7.1: The basic operations of the stack ADT.

Only `push` and `pop` change the stack, adding or removing one item; the other three only report. `top()` and `pop()` both return the top item, but `top()` leaves it in place. When you must look at an item before deciding whether to remove it (as in the next greater element problem, 7.3.9), call `top()` and then `pop()` .

On an empty stack there is no top item, so `pop()` and `top()` have nothing to return. Popping an empty stack is an error called **underflow**. Pushing onto a full stack of limited capacity is called **overflow**, which can only happen to bounded stacks (7.2.1).

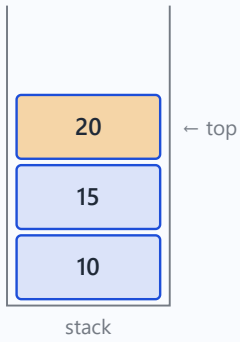
7.1.3 push and pop, step by step

Starting from an empty stack, we apply `push(10)` , `push(15)` , `push(20)` , `top()` , and four `pop()` calls.

Example 7.1

1. **Start.** The stack is empty: `size = 0` .
2. `push(10)` : As the only item, 10 is both bottom and top. `size = 1` .
3. `push(15)` : 15 goes on 10 and becomes the top. `size = 2` .
4. `push(20)` : Bottom to top, the stack is 10, 15, 20. `size = 3` .
5. `top()` : 20 is returned but not removed. `size` stays 3.
6. `pop()` : 20, last in, comes out first. The top is 15; `size = 2` .
7. `pop()` : 15 comes out. The top is 10; `size = 1` .
8. `pop()` : 10 comes out; the stack is empty, `size = 0` .
9. `pop()` : Nothing to remove: underflow, reported as an error.

The returned values are 20 (`top`), 20, 15, 10, and an error. The items left in the order 20, 15, 10, the reverse of their arrival.



Operation	Returns	size
`push(10)`	–	1
`push(15)`	–	2
`push(20)`	–	3

underflow!



Operation	Returns	size
`push(10)`	–	1
`push(15)`	–	2
`push(20)`	–	3
`top()`	20	3
`pop()`	20	2
`pop()`	15	1
`pop()`	10	0
`pop()`	error!	0

Figure 7.1: Two moments from Example 7.1. Top: after three `push` calls the stack holds 10, 15, 20 bottom to top, and top points to 20. Bottom: three `pop` calls have emptied the stack, and the fourth underflows. The tables give each operation's return value and the `size` after it.

The table in Figure 7.1 is a handy layout for stack exercises: one row per operation, with its return value and the size afterward. Writing down the stack itself, bottom to top, at each step keeps you from popping the wrong item.

Common mistake

Believing that `top()` changes the stack. It only reads: above, the first `pop()` after `top()` still returns 20. Had `top()` removed the item, that `pop()` would return 15.

7.1.4 A Java interface: separating the ADT from its implementation

In Java an ADT is naturally expressed as an **interface**, which lists the methods a class must provide without giving their bodies. For the stack:

```
public interface Stack<E> {
  int size();
  boolean isEmpty();
  void push(E e);
  E top();    // null if empty
  E pop();    // exception if empty
}
```

The `<E>` makes the stack **generic**. `E` is a type parameter: write `Stack<Integer>` for integers or `Stack<Character>` for characters, and `E` stands for that type throughout, so one piece of code serves every item type. The methods are those of Table 7.1, and the comments make empty-stack behavior part of the contract: `top()` returns `null`, while `pop()` throws an **exception**.

The interface separates "what" from "how":

- **The ADT answers "what?"** It names the operations and their meaning and says nothing about arrays or linked lists.
- **The implementation answers "how?"** A class such as `ArrayStack<E> implements Stack<E>` writes each method for one storage scheme; a linked-list class can implement the same interface.
- **Code written against the interface can swap implementations.** A method taking a `Stack<Integer>` accepts an array-based or a linked stack without a line of the caller changing.

This matters in practice: if measurements call for a faster implementation, the interface keeps the change local.

Empty-stack behavior, however, differs between libraries:

On an empty stack	`pop()`	`top()` / `peek()`
Our own `Stack<E>`	throws an exception	returns `null`
`java.util.Stack`	`EmptyStackException`	`EmptyStackException`
`java.util.ArrayDeque`	`NoSuchElementException`	returns `null`

Table 7.2: Behavior of `pop()` and `top()` / `peek()` on an empty stack.

Because of the differences in Table 7.2, always check `isEmpty()` before calling `pop()` or `peek()`. Otherwise the same code throws with one library and silently gets `null` with another, and the bug surfaces much later, where it is hard to trace.

Common mistake

Using our `Stack` interface and `java.util.Stack` in the same file. With `import java.util.*;` or `import java.util.Stack;`, the compiler may find the name ambiguous or pick the class you did not mean. If you use our interface, do not import `java.util.Stack`.

7.1.5 Algorithms for the operations

Before implementing, we write the operations in language-neutral pseudocode. Assume the stack lives in an array, along with an integer `top` holding the *index* of the top item. Indices start at 0, so the first item goes into cell 0, and an empty stack is represented by `top = -1`.

push(x):

```
begin
  if the stack is full   // top = cap-1
    error: overflow
  else
    increment top by 1
    stack[top] ← x
end
```

With capacity `cap`, the last valid index is `cap - 1`; once `top` reaches it, the array is full and a push overflows. Otherwise `top` is incremented and `x` is written into the new top cell. The order matters: done the other way round, `x` would overwrite the current top item.

pop():

```
begin
  if the stack is empty  // top = -1
    error: underflow
  else
    x ← stack[top]; decrement top by 1
    return x
end
```

If `top = -1`, we have underflow. Otherwise the top item is saved, `top` is decremented, and the saved item is returned; again, read first, then decrement.

top():

```
begin
  if the stack is empty return null
  else return stack[top]
end
```

isEmpty() and size():

```
isEmpty: true if top = -1, else false
size:    return top + 1
```

Why `top + 1`? The items occupy indices 0 through `top`, which is `top + 1` indices. A three-item stack uses cells 0, 1, 2, so `top = 2` and `size = 3`; an empty stack has `top = -1` and `size = 0`.

> **Common mistake**
>
> Confusing `top` with `size`. `top` is the index of the last item, not the item count. The mix-up causes two typical bugs: testing fullness with `top == cap` instead of `top == cap - 1`, and having `size()` return `top` instead of `top + 1`. With the first, one more push is attempted on a full array, and `stack[cap]` falls outside it.

7.1.6 Time and space cost

The stack's great strength is that all its basic operations take constant time. An operation whose running time does not depend on the number of items takes **O(1)** time. Table 7.3 gives the costs and their reasons.

Operation	Time	Why?
`push`	O(1)	only the top changes
`pop`	O(1)	only the top changes
`top`	O(1)	the top is known directly
`isEmpty`, `size`	O(1)	a counter or index is kept
searching for an item	O(n)	items must be popped one by one

Table 7.3: Time cost of stack operations (n: number of items on the stack).

The pseudocode of 7.1.5 shows why. Each operation does at most one comparison, one index update, and one array access, and accessing a known index takes the same time whatever the array's size. There are no loops; nothing is shifted or scanned. A `push` onto a million-item stack takes as many steps as onto a three-item one.

Searching is different. The ADT's only way to find a value is to pop items from the top and look at each. In the worst case (the value is at the bottom or absent) all *n* items come off: *O(n)*. If you search often, a stack is the wrong structure.

A stack of *n* items must store them all, so it uses *O(n)* memory, while each operation needs only a few local variables: *O(1)* extra space.

There is one exception. In a stack built on a **dynamic array**, which grows when full, the `push` that triggers growth copies every item into a new array and takes *O(n)*. Growth is rare, so the average cost per `push` over a long sequence is still constant, or **amortized O(1)**; Section 7.2.6 works it out.

The speed comes from the restriction itself: with access limited to the top, no operation ever shifts or searches. A list that allows insertion in the middle or reading any index is more flexible, but pays with operations that take *O(n)*.

7.2 Implementation

To **implement** an ADT is to write it as a class built on a particular storage scheme. For a stack there are two basic options, an array or a list of linked nodes. This section codes both and compares them.

7.2.1 Fixed-size and dynamic-size stacks

First decide how many items the stack can hold; the two answers give two kinds of stack.

A **fixed-size stack** sets its capacity at creation and never grows or shrinks; underneath is a fixed-length array. Pushing onto a full stack is an overflow error, popping an empty one an underflow error.

A **dynamic-size stack** grows and shrinks at run time. When full it raises its own capacity, say by doubling, and when the item count falls far below capacity it may shrink to return memory. It is built on a dynamic array or a linked list.

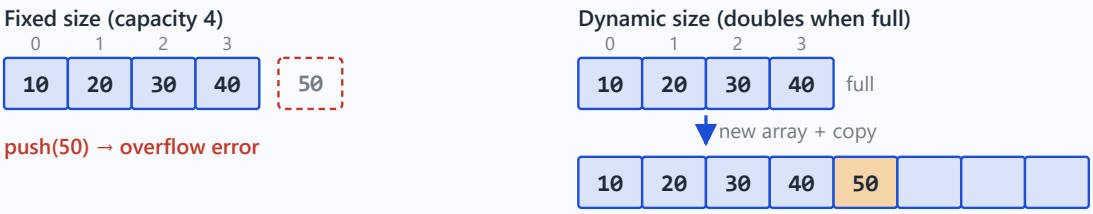


Figure 7.2: Fixed-size versus dynamic-size stacks. Left: an array of capacity 4 is full (10, 20, 30, 40), and `push(50)` overflows. Right: a dynamic stack in the same situation allocates an array of twice the capacity (8 cells), copies the old items, and writes 50 at index 4 of the new array.

The two differ only when full. A dynamic stack can grow, but popping an empty one is still meaningless, so overflow occurs only in a fixed-size stack while underflow can occur in any.

7.2.2 Two ways to implement a stack

The stack ADT can be implemented with an **array** or a **linked list**. Figure 7.3 shows the same stack (10, 15, 20 bottom to top) both ways.

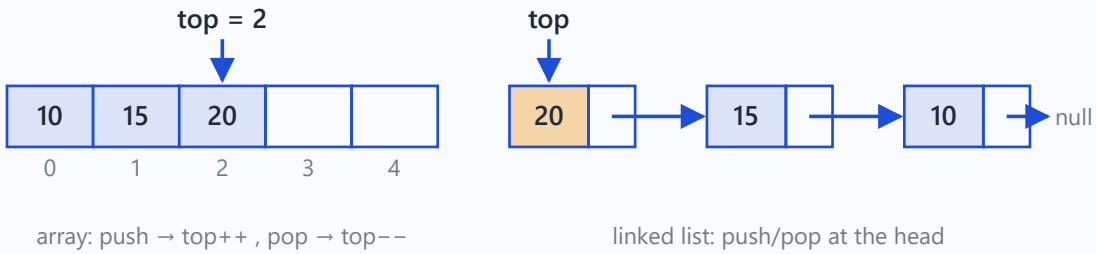


Figure 7.3: Two implementations of one stack. Left, array-based: the items are at indices 0, 1, 2 and `top = 2`; `push` increments `top`, `pop` decrements it. Right, linked: `top` points to node 20, the chain is 20 → 15 → 10 → null, and `push` and `pop` work at the head of the list.

In the array-based stack, `push` increments `top` and writes the item into `arr[top]`; `pop` takes `arr[top]` and decrements `top`. The bottom is always cell 0, and the stack grows toward the end of the array.

In the linked stack, the top is the list's **head**: `push` inserts a node there, and `pop` removes it and returns its value. The pictures differ, yet as ADTs the stacks are identical: both have 20 on top, and a `pop` returns 20 from either.

7.2.3 The array-based stack

An array-based stack whose capacity is fixed at creation is also called a **bounded stack**. Figure 7.4 shows three items (5, 8, 3) in an array of capacity 8.

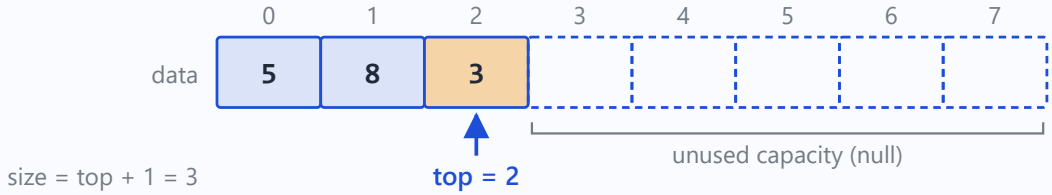


Figure 7.4: The array `data`, of capacity 8, holds 5, 8, 3. `top = 2` marks the last item, so `size = top + 1 = 3`. Cells 3–7 are unused capacity and contain `null`.

Its advantages:

- **Simplicity.** An array and an integer suffice; the code is a few lines.
- **Memory economy.** Only the items are stored, with no per-node reference as in a linked list.

- **Strict memory control.** The stack can never take more than its capacity, a safeguard against malicious input that tries to grow a stack until memory runs out.
- **Cache friendliness.** The items occupy consecutive cells, and since the processor reads memory in blocks, consecutive data is faster to reach than scattered nodes. The same *O(1)* operations thus usually run faster in practice.

Points to watch:

- Check for fullness before pushing; otherwise the write falls outside the array and Java throws `ArrayIndexOutOfBoundsException`.
- Capacity is a compromise: too small and the stack overflows, too large and idle cells waste memory.
- In an array of objects, set each vacated cell to `null`, or the array keeps the removed object alive in memory for nothing.

7.2.4 Code: ArrayStack

This class implements the `Stack<E>` interface of 7.1.4 with a fixed-capacity array.

```
public class ArrayStack<E> implements Stack<E> {
    private E[] data;
    private int top = -1;                    // -1 → empty stack
    public ArrayStack(int capacity) { data = (E[]) new Object[capacity]; }
    public boolean isEmpty() { return top == -1; }
    public int size()        { return top + 1; }
    public E top()           { return isEmpty() ? null : data[top]; }
    public void push(E x) {
        if (top == data.length - 1) throw new IllegalStateException("Overflow");
        data[++top] = x;
    }
    public E pop() {
        if (isEmpty()) throw new IllegalStateException("Underflow");
        E x = data[top];
        data[top--] = null;                  // drop the reference
        return x;
    }
}
```

How it works:

- `data` holds the items; `top`, the index of the top item, starts at `-1` because a new stack is empty.
- The constructor allocates the array. Java forbids `new E[capacity]`, because the type parameter `E` is unknown at run time, so we create an `Object` array and cast it with `(E[])`. The compiler warns that the cast is “unchecked”; it is safe as long as the array never leaves the class.
- `isEmpty()` tests `top == -1`; `size()` returns `top + 1` (7.1.5).

- `top()` returns `null` for an empty stack, else `data[top]`. The conditional operator `? :` yields its first value if the condition holds, its second otherwise. The class has both a field `top` and a method `top()`; Java tells them apart by the parentheses.
- `push(x)` throws `IllegalStateException("Overflow")` if `top == data.length - 1`, meaning the last cell is taken. Otherwise `data[++top] = x;` runs. `++top` is a *pre-increment*: `top` rises first, and the new value serves as the index, fitting "increment, then write" (7.1.5) on one line.
- `pop()` throws an "Underflow" exception on an empty stack. Otherwise it saves the top item in `x`. In `data[top--] = null;`, `top--` is a *post-decrement*: the old `top` is used as the index and that cell set to `null`, and then `top` drops by one. Finally `x` is returned.

`++top` versus `top++` is decisive: `data[top++] = x` would overwrite the current top item, and `data[--top] = null` would clear the cell below the top.

No method loops, so each runs in *O(1)* time. The array has one cell per unit of capacity: *O(capacity)* space.

> **Common mistake**
>
> Not clearing the vacated cell. If `pop` only does `top--`, it returns the right value but the array still refers to the removed object. Java's **garbage collector** frees only objects nothing refers to, so an object the stack has finished with but the array still references cannot be reclaimed. This kind of memory leak is called *loitering*. Arrays of primitives such as `int[]` hold no references and are immune.

7.2.5 Code trace: ArrayStack

We trace a stack of capacity 4 as `main` calls `s.push(10)`, `s.push(15)`, `s.push(20)`, `s.top()`, `s.pop()`, `s.push(25)`, `s.push(30)`, `s.push(40)`. The method bodies involved:

```
public void push(E x) {
  if (top == data.length - 1)
    throw new IllegalStateException("Overflow");
  data[++top] = x;
}
public E pop() {
  if (isEmpty())
    throw new IllegalStateException("Underflow");
  E x = data[top];
  data[top--] = null;          // drop the reference
  return x;
}
public E top() { return isEmpty() ? null : data[top]; }
```

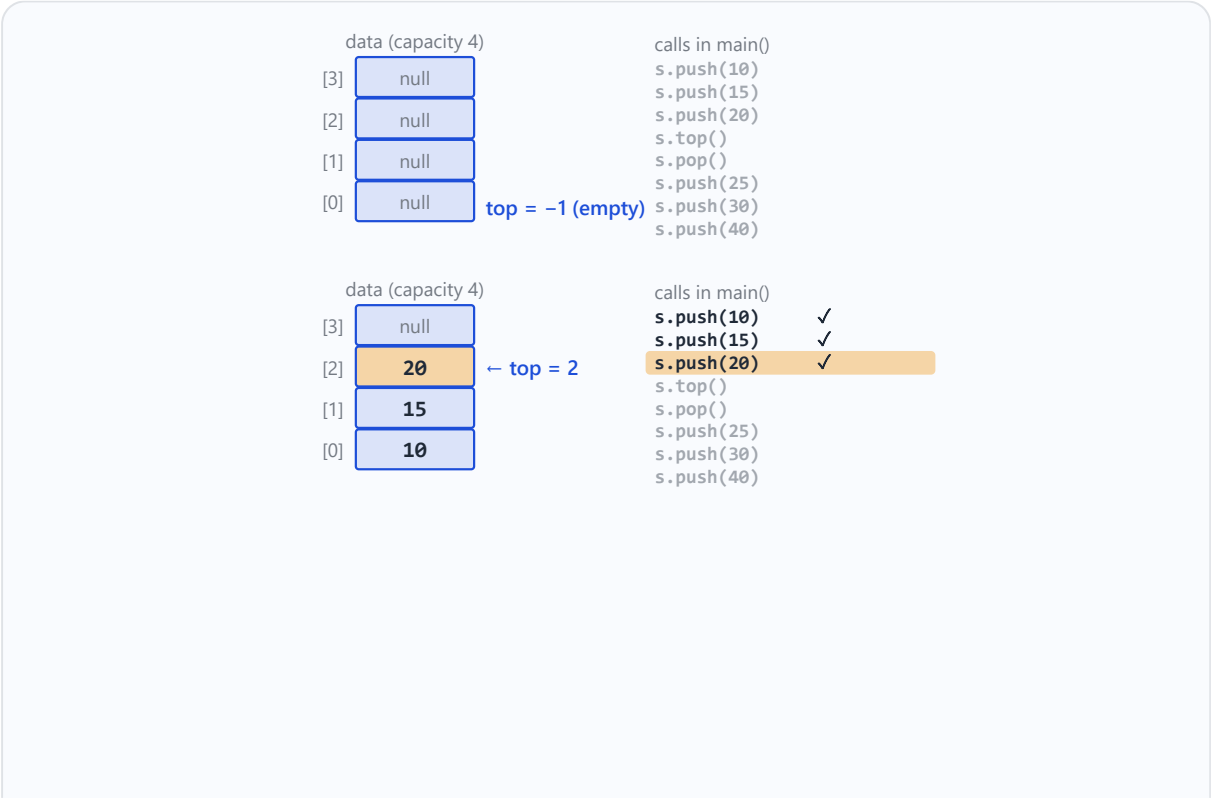
Call	Full/empty check	`top`	`data`	Returns
(start)	—	−1	[null, null, null, null]	—
`push(10)`	−1 == 3? no	0	[10, null, null, null]	—
`push(15)`	0 == 3? no	1	[10, 15, null, null]	—
`push(20)`	1 == 3? no	2	[10, 15, 20, null]	—
`top()`	empty? no	2	[10, 15, 20, null]	20
`pop()`	2 == −1? no	1	[10, 15, null, null]	20
`push(25)`	1 == 3? no	2	[10, 15, 25, null]	—
`push(30)`	2 == 3? no	3	[10, 15, 25, 30]	—
`push(40)`	3 == 3? **yes**	3	[10, 15, 25, 30]	exception: “Overflow”

Table 7.4: Trace of the call sequence on `new ArrayStack<>(4)`. `data` is written from index 0 to 3.

On the first `push(10)`, `top = −1` differs from `data.length − 1 = 3`, so `data[++top] = x` raises `top` to 0 and writes `data[0] = 10`, one assignment regardless of array size.

`top()` returns `data[2] = 20`, changing nothing. The following `pop()` finds `top = 2 ≠ −1`, saves `x = data[2] = 20`, clears cell 2 with `data[top--] = null`, and lowers `top` to 1. It returns 20: last in, first out.

`push(25)` reuses cell 2, and `push(30)` fills the last cell, so `top = 3`. At `push(40)` the check yields `3 == 3`: the array is full, `IllegalStateException("Overflow")` is thrown, and the array and `top` stay as they were. That is overflow in a fixed-capacity stack; a dynamic array would grow here instead.



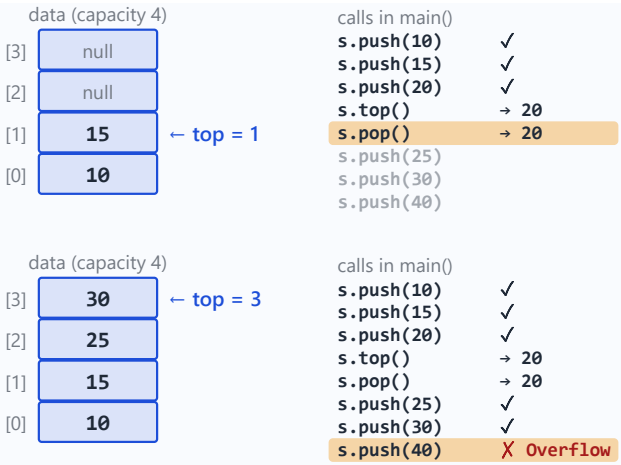


Figure 7.5: Four moments of the trace (left to right, top to bottom). (a) Start: four `null` cells, `top = -1`. (b) After three `push` calls, `top = 2`. (c) `pop()` has returned 20; cell 2 is `null`, `top = 1`. (d) After `push(25)` and `push(30)` the array is full, `top = 3`; `push(40)` overflows.

In frames (c) and (d) of Figure 7.5, cell 2 holds 20, becomes `null` after the `pop`, and is refilled with 25. An array-based stack reuses its cells; removing an item never shifts the rest.

7.2.6 Dynamic arrays: doubling when full

A fixed capacity fails when the number of items is unknown in advance. The remedy is to move the items to a bigger array when the current one fills; since a Java array cannot change length, "growing" means creating a new array and copying.

```
public void push(E x) {
  if (top == data.length - 1)
    resize(2 * data.length);
  data[++top] = x;
}
private void resize(int newCap) {
  E[] bigger = (E[]) new Object[newCap];
  for (int i = 0; i <= top; i++)
    bigger[i] = data[i];
  data = bigger;
}
```

Instead of throwing, the new `push` calls `resize(2 * data.length)`, which allocates the larger array, copies the items at indices 0 to `top` to the same indices, and points `data` at the new array. The old array is now unreferenced, and the garbage collector reclaims it. Then `data[++top] = x` runs as usual.

A resizing `push` copies n items and so takes $O(n)$. But resizes are rare: since the capacity doubles each time, a resize is followed by as many cheap pushes as there were items before it. The cost of the expensive operations is spread over the cheap ones in between.

Definition 7.2

The **amortized cost** of an operation is the total cost of a long sequence of operations divided by their number. Unlike the worst-case cost of a single operation, it averages occasional expensive operations with the cheap ones. No probability is involved: it is an upper bound valid for *every* sequence. Finding such a bound is called **amortized analysis**.

Example 7.2

We push 17 times onto an empty dynamic-array stack of capacity 4. Each `push` does one write, and a resizing `push` also copies all current items. How many copies are made, and what is the total cost?

1. **Pushes 1–4:** Writes only. After the 4th the array is full.
2. **Push 5:** `top == 3 == data.length - 1`; capacity goes from 4 to 8 and 4 items are copied. Then the 5th item is written.
3. **Pushes 6–8:** Writes only.
4. **Push 9:** The array is full with 8 items; capacity becomes 16, and 8 items are copied.
5. **Pushes 10–16:** Writes only.
6. **Push 17:** 16 items are copied; capacity becomes 32.
7. **Total.** Copies: 4 + 8 + 16 = 28. Writes: 17. Total 45 basic steps, below the bound $3n = 51$: on average 45 / 17 ≈ 2.65 steps per `push`.

This generalizes. The copy counts 4, 8, 16 each double the previous one, and such a sum is less than twice its last term. The last resize copies fewer than n items, so all copies together stay below $2n$.

Theorem 7.1 (Amortized cost of doubling)

In a dynamic-array stack whose capacity doubles when full, n pushes starting from an empty stack cost less than $3n$ in total. Hence `push` has amortized cost $O(1)$.

Proof. Let the initial capacity be 1 (a larger one only reduces copying). Resizes happen when the array fills with 1, 2, 4, ..., 2^k items, where 2^k is the largest power of two below n. The copies number $1 + 2 + 4 + \ldots + 2^k = 2^{k+1} - 1$, and since $2^k < n$, this is less than $2n - 1$, hence less than $2n$. Each `push` also does one write, n writes in all. The total is $< 2n + n = 3n$, an average of less than 3 per `push`, a constant. □

The theorem does not make every `push` fast: the 17th `push` of Example 7.2 made 16 copies. In a game or real-time system where every operation must be quick, only the worst case counts.

Growing by a fixed amount instead of a factor is far worse. Let the capacity start at 10 and grow by 10 each time it fills (10 → 20 → 30 → ...). Resizes happen at 10, 20, 30, ... items, and the copies total *10 + 20 + 30 + ...* . For *n = 100* that is *10 + 20 + ... + 90 = 450*, and in general about $n^2/20$. The total cost is $O(n^2)$, or *O(n)* per `push` on average. For 1000 pushes, doubling makes 1023 copies; growing by 10 makes 49,500.

> **Common mistake**
>
> Equating "amortized *O(1)*" with "*O(1)* per operation." In a dynamic array, `push` is amortized *O(1)*, but a resizing `push` is *O(n)*. A second mistake is growing the capacity by a fixed amount, which raises even the amortized cost to *O(n)*. Grow by a factor (for example ×2).

7.2.7 The linked-list stack

The second implementation keeps items in a chain of linked **nodes**. Each node has two fields: the item's value and a reference to the next node.

```
class Node {
    int data;
    Node next;
    Node(int data) { this.data = data; }   // next = null
}
```

In `Node`, `data` is the value and `next` refers to the node below. The constructor sets only `data`; unassigned reference fields default to `null` in Java, so a new node's `next` is `null`.

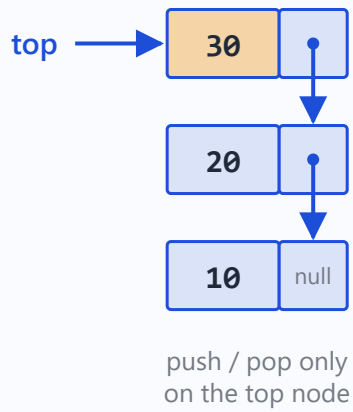


Figure 7.6: A stack as a linked list. `top` points to 30, 30 to 20, and 20 to 10, whose `next` is `null`. `push` and `pop` work only on the head (top) node.

Properties of this implementation:

- **Flexible size.** Each `push` allocates a node, so the size adapts at run time; there is no overflow until memory runs out.
- **Extra memory.** Each node stores a `next` reference besides its value. For small items such as an `int`, the reference can take as much space as the item or more.
- **No random access, but a fast top.** Reaching the *k*th item takes *k* steps from the head. A stack, though, needs only its top, which is the head, and the `top` reference points straight at it.
- **Insertion and removal in *O(1)*.** Both happen at the head and change only a few references.

Why the head and not the tail? In a **singly linked list** each node knows only its successor. Deleting the last node means setting the second-to-last node's `next` to `null`, and reaching that node requires walking from the head: *O(n)*. Deleting the head merely advances the head reference: *O(1)*. With the top at the head, both operations are constant-time.

7.2.8 Code trace: push on a linked list

The linked stack's fields and `push` method:

```
Node top;          // top of the stack
int size;

public void push(int data) {
  Node temp = new Node(data);
  temp.next = top;
  top = temp;
  size++;
}
```

`top` refers to the top node and `size` counts the items; an empty stack has `top = null` and `size = 0`. `push` takes four steps: create the node, held by the local variable `temp`; link its `next` to the old top; point `top` at it; increment `size`. We trace `push(10)` and then `push(15)` on an empty stack.

Line	`temp`	`top`	Chain	`size`
(start)	—	null	null	0
`temp = new Node(10)`	node(10), next = null	null	null	0
`temp.next = top`	node(10) → null	null	null	0
`top = temp`	node(10)	node(10)	10 → null	0
`size++`	(method returns)	node(10)	10 → null	1
`temp = new Node(15)`	node(15), next = null	node(10)	10 → null	1
`temp.next = top`	node(15) → node(10)	node(10)	10 → null	1
`top = temp`	node(15)	node(15)	15 → 10 → null	1

Line	temp	top	Chain	size
size++	(method returns)	node(15)	15 → 10 → null	2

Table 7.5: Trace of `push(10)` and `push(15)` on an empty stack. The "Chain" column lists the nodes starting from `top`.

On the first `push(10)`, `top` is `null`, so `temp.next = top` leaves `next` `null` and the node stands alone. When the method returns, the local `temp` vanishes but the node survives, now reachable through `top`.

On the second `push`, node 15 is not yet in the list when created; `top` still points to 10. `temp.next = top` links 15 to the old top, and only then does `top = temp` move the top. Nothing is shifted and just two references change, so `push` is *O(1)*.

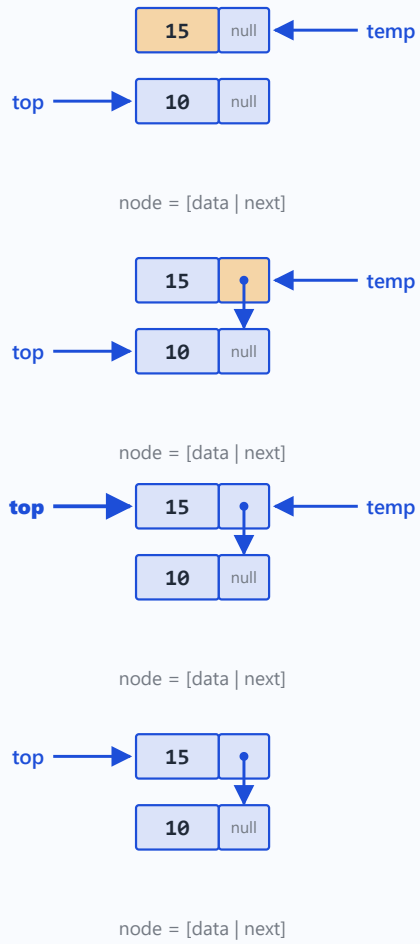


Figure 7.7: The steps of `push(15)`. (a) Node 15 is created and `temp` points to it; `top` is still at 10. (b) `temp.next = top`: 15 is linked to 10. (c) `top = temp`: 15 is the new top. (d) The method has returned: top to bottom, 15 → 10.

Common mistake

Swapping the two lines in `push`, so that `top = temp` runs before `temp.next = top`. Then `top` already refers to the new node, and the second line points the node's `next` at itself. The old list, no longer referenced, is lost, and the stack becomes one node pointing to itself. Always link the new node to the old list first, then move the top.

7.2.9 Code trace: pop on a linked list

The `pop` method, and a loop calling it three times:

```
public int pop() {
  if (top == null)
    throw new RuntimeException("empty");
  int result = top.data;
  top = top.next;
  size--;
  return result;
}
// in main (stack: 15 → 10):
for (int k = 0; k < 3; k++)
  System.out.println(pop());
```

If `top == null`, `pop` throws `RuntimeException("empty")`. Otherwise it saves the top value in `result`, advances `top` to the node below, decrements `size`, and returns the value. The value must be saved *before* the node is unlinked: once `top` moves, the old top node is unreachable.

Call	`top == null` ?	`result`	Chain (after)	`size`	Printed
1st `pop()`	no	15	10 → null	1	15
2nd `pop()`	no	10	null	0	10
3rd `pop()`	**yes**	—	null	0	exception: "empty"

Table 7.6: The loop's three `pop()` calls on the stack 15 → 10.

On the first call, `top = top.next` moves the top down to node 10. Nothing refers to the old node 15 any more, and the garbage collector will reclaim it. No explicit `null` is needed, unlike in the array stack, because `top` was the node's only reference and it has moved on. The second call sets `top` to `null`, emptying the stack. The third finds the stack empty and throws, and since nothing catches the exception, the program stops.

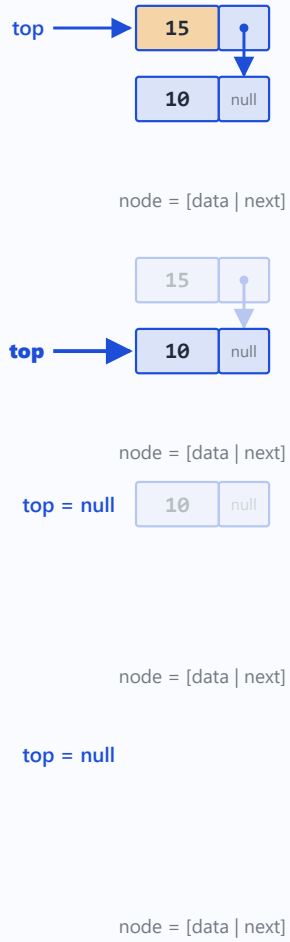


Figure 7.8: The steps of `pop()`. (a) `result = top.data`: 15 is saved. (b) `top = top.next`: the top moves to 10; the faded node 15 is unreferenced. (c) On the second `pop`, `top` becomes `null`; the stack is empty. (d) The third `pop` finds an empty stack: underflow exception.

Without the emptiness check, `top.data` would dereference `null` on the third call and throw `NullPointerException`. The program stops either way, but a descriptive exception names the cause, not just a symptom.

7.2.10 Array or linked list?

Both implementations perform the basic operations in *O(1)*, so the choice rests on secondary criteria such as memory use and capacity behavior.

Criterion	Array-based	Linked-list-based
Size	fixed capacity (grown in a dynamic array)	flexible, changes at run time
`push` / `pop`	O(1) (amortized O(1) with resizing)	O(1)
Overflow	possible	only when memory runs out
Memory per item	data only	data + `next` reference
Wasted space	unused capacity	none

Criterion	Array-based	Linked-list-based
Java	`ArrayDeque`, `Stack` (based on `Vector`)	`LinkedList` (as a `Deque`)

Table 7.7: Array-based versus linked-list-based stacks.

The table shows a trade-off. The array uses less memory per item, but its unused capacity may be wasted; the linked list keeps no spare capacity but carries a reference per item. A dynamic array occasionally copies *O(n)* items in one `push`, while a linked list allocates an object on every `push`, a constant but not negligible cost. Thanks to the cache, the array-based stack is usually faster in practice.

For new Java code, the recommended stack is the `Deque` interface with the `ArrayDeque` class:

```
Deque<Integer> s = new ArrayDeque<>();
```

It provides `push`, `pop`, and `peek`. A `Deque` (double-ended queue) allows insertion and removal at both ends; as a stack it uses only the front. The old `java.util.Stack` extends `Vector`, whose methods are all synchronized: each call takes a lock against simultaneous access by several threads, a needless slowdown in single-threaded programs. Inheritance from `Vector` also exposes operations foreign to a stack, such as inserting at an arbitrary index. The rest of these notes therefore use `ArrayDeque`.

7.3 Stack applications

The problems in this section look unrelated: browser history, function calls, parentheses, number sequences. Yet each asks one question, “Which item, seen most recently, is still unresolved?”, and a stack answers it in constant time.

7.3.1 Where stacks are used

Stacks appear throughout computer science:

- **Expressions.** Converting between infix, prefix, and postfix, evaluating postfix, and checking parentheses (7.5 and 7.3.5).
- **Undo / redo** in text editors.
- **Browser history.** “Back” and “forward” are managed with two stacks.
- **The call stack.** Function calls, local variables, and return addresses.
- **Backtracking.** In maze solving, n-queens, or the Knight’s Tour, returning to the latest decision point after a dead end.
- **Algorithms.** Towers of Hanoi, tree traversal, stock span, histogram problems.
- **Graph algorithms. Depth-first search** (DFS), topological sorting, strongly connected components.

- **String reversal.** Characters go in in order and come out reversed.
- **Next greater element.** For each item of an array, the first larger value to its right.

In each, jobs start nested and finish in reverse order: the latest decision is undone first, the last bracket opened closes first, the last method called returns first. Such nesting is exactly what LIFO models.

7.3.2 Back and forward in a browser: two stacks

A browser runs "back" and "forward" on two stacks. The *back stack* holds pages visited before the current one, the *forward stack* pages left by going back; the current page sits in its own variable. The operations:

- `visit(s)`: push the current page onto the back stack, clear the forward stack, and make `s` current.
- `back()`: push the current page onto the forward stack; the page popped from the back stack becomes current.
- `forward()`: push the current page onto the back stack; the page popped from the forward stack becomes current.

Example 7.3

The browser opens on "Home," followed by `visit(News)`, `visit(Sports)`, `back()`, `back()`, `forward()`, `visit(Weather)`, `back()`. The table gives the state after each step (stacks written bottom to top).

Operation	Back stack	Current page	Forward stack
(open)	[]	Home	[]
`visit(News)`	[Home]	News	[]
`visit(Sports)`	[Home, News]	Sports	[]
`back()`	[Home]	News	[Sports]
`back()`	[]	Home	[Sports, News]
`forward()`	[Home]	News	[Sports]
`visit(Weather)`	[Home, News]	Weather	[] (Sports deleted)
`back()`	[Home]	News	[Weather]

Table 7.8: Browser history with two stacks.

The key step is `visit(Weather)`. Sports is on the forward stack, but a new link from News starts a new path, so Sports is no longer "forward," and the forward stack is cleared.

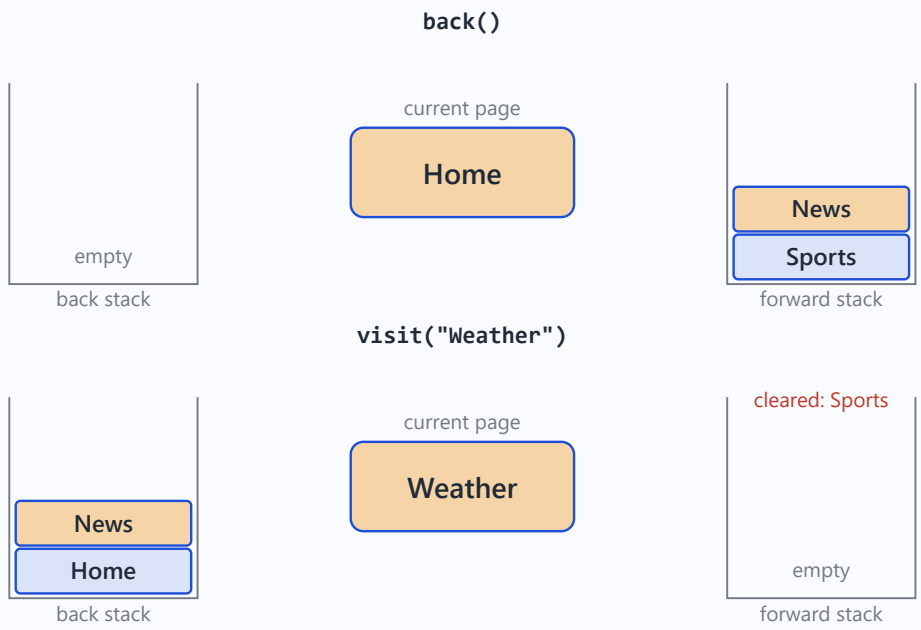


Figure 7.9: Two moments from Example 7.3. Top, after two `back()` calls: the back stack is empty, the current page is Home, and the forward stack holds Sports and News, bottom to top. Bottom, after `visit(Weather)`: the back stack holds Home and News, and the forward stack has been cleared (Sports deleted).

Every operation is *O(1)* except clearing the forward stack, and even that is *O(1)* if we swap in a fresh empty stack. Undo/redo works the same way: the undo stack holds changes, the redo stack undone ones, and a new change clears the redo stack.

> **Common mistake**
>
> Forgetting to clear the forward stack on a new visit. “Forward” from Weather would then lead to Sports, on an abandoned path.

7.3.3 The call stack

Every method call gets a record in memory holding its parameters, its local variables, and where execution resumes afterward (the return address). This **stack frame** lives on the **call stack**: a call pushes it, a return pops it, and control passes to the caller, now on top.

A method that calls itself, which is called **recursion**, shows the call stack most clearly. This one computes $n! = n \cdot (n - 1) \cdot \ldots \cdot 1$.

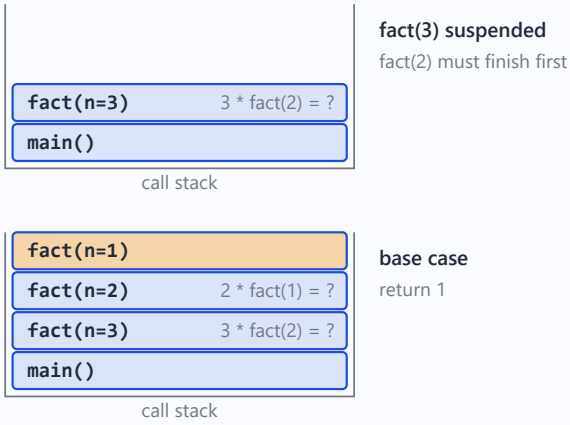
```
int fact(int n) {
  if (n <= 1)
    return 1;
  return n * fact(n - 1);
}
// in main:
System.out.println(fact(3));
```

The case `n <= 1`, answered directly without a recursive call, is the **base case**. Every other case computes `n * fact(n - 1)`, and the multiplication must wait until `fact(n - 1)` finishes.

Step	Event	Call stack
1	`main` runs	main
2	`fact(3)` called; 3 <= 1 false; `3 * fact(2)` waits	main, fact(3)
3	`fact(2)` called; 2 <= 1 false; `2 * fact(1)` waits	main, fact(3), fact(2)
4	`fact(1)` called; 1 <= 1 true; `return 1`	main, fact(3), fact(2), fact(1)
5	`fact(1)` popped, returned 1; `fact(2)` : 2 · 1 = 2	main, fact(3), fact(2)
6	`fact(2)` popped, returned 2; `fact(3)` : 3 · 2 = 6	main, fact(3)
7	`fact(3)` popped, returned 6; `main` prints 6	main

Table 7.9: The call stack during `fact(3)` (bottom to top).

The stack grows in steps 2–4 and shrinks in steps 5–7; this shrinking after the base case is called **unwinding**. Each suspended frame resumes when the frame above it returns: `fact(2)` finishes its multiplication with the 1 from `fact(1)`.



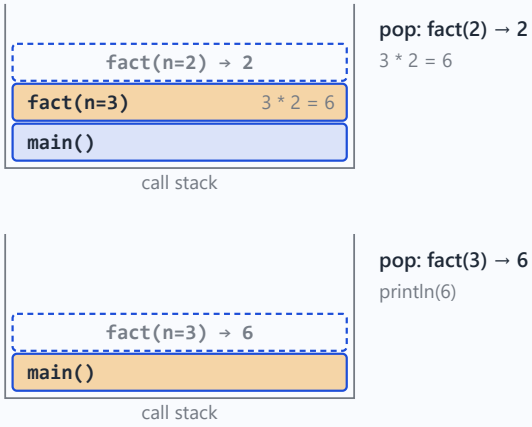


Figure 7.10: The call stack in `fact(3)`. (a) `fact(3)` is suspended: `3 * fact(2)` cannot be computed yet. (b) Base case: `fact(1)` is on top and will return 1. (c) `fact(2)` has been popped, returning 2; `fact(3)` computes 3 · 2. (d) `fact(3)` has returned 6; only `main` remains.

The call stack is at most n frames deep, so `fact(n)` uses $O(n)$ memory. Its size is limited: very deep recursion, or recursion that never reaches its base case, fills it, and Java throws `StackOverflowError`, an *overflow* of the call stack.

Common mistake

Omitting the base case, or making it unreachable (calling `fact(n + 1)`, say). Every call pushes a frame and none returns, until the program dies with `StackOverflowError`.

7.3.4 Reversing a string

Items pushed in order come out reversed, which is all we need to reverse a string.

```
String reverse(String str) {
    Deque<Character> stack = new ArrayDeque<>();
    char[] chars = str.toCharArray();
    for (char c : chars)
        stack.push(c);
    for (int i = 0; i < str.length(); i++)
        chars[i] = stack.pop();
    return new String(chars);
}
// in main:
System.out.println(reverse("ABCD"));
```

The stack is a `Deque<Character>` because Java generics take wrapper classes (`Character`), not primitives (`char`); `push(c)` converts automatically. `toCharArray()` copies the characters into a new array, and since a `String` is immutable, the result is built by overwriting that array.

The first loop pushes the characters; the second refills the array from `i = 0` with popped ones. For `"ABCD"` the stack becomes A, B, C, D bottom to top; the first `pop` gives D, so `chars[0] = 'D'`, then C, B, A follow. The result is `"DCBA"`.

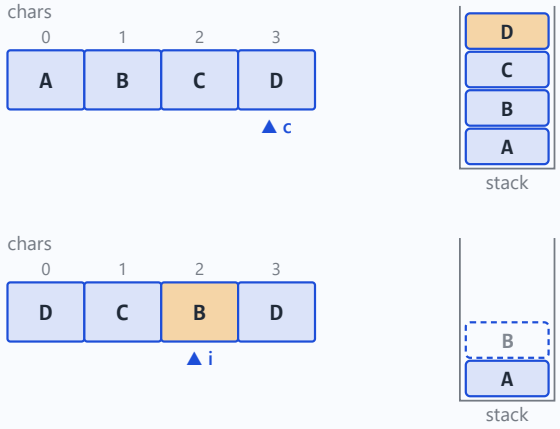


Figure 7.11: `reverse("ABCD")`. (a) After the first loop: the stack is A, B, C, D bottom to top, with the last character on top. (b) Second loop, `i = 2`: D and C are in place, and B is popped into cell 2.

Each character is pushed and popped once: *O(n)* time, and the stack and array need *O(n)* extra memory. The same idea tests for a **palindrome**, a word that reads the same backward ("kayak"): compare the reversed string with the original.

7.3.5 Balanced parentheses: the problem and the algorithm

One of a compiler's first jobs is checking that brackets match. Simplified, let the string `S` consist only of `( ) { } [ ]`.

Definition 7.3

A string `S` over the characters `( ) { } [ ]` is **valid** if and only if:

1. every opening bracket is closed by a closing bracket of the same type;
2. brackets are closed in the right order: the last one opened closes first.

This check is called **balanced parentheses**.

The second condition is the LIFO rule itself, so the algorithm keeps the *unmatched* opening brackets on a stack and scans `S` once:

1. An **opening bracket** is pushed to await its closing partner.
2. For a **closing bracket**: if the stack is empty, `S` is invalid (nothing to close). Otherwise pop; if the popped bracket is not of the same type, `S` is invalid.

3. **At the end of** `S`, it is valid if the stack is empty and invalid otherwise (some bracket was never closed).

The kinds of error it catches:

String	Result	Why?
`{[()]}`	valid	each bracket closes in reverse order
`([)]`	invalid	wrong order: when `)` arrives, the latest open is `[`
`(()`	invalid	unclosed: `(` remains on the stack
`())`	invalid	extra closer: the last `)` meets an empty stack

Table 7.10: Bracket strings and results.

`([)]` shows why counting is not enough: one opener and one closer of each type, so the counts balance, but the order is wrong. A counter misses this; a stack does not.

7.3.6 Code: isBalanced()

```
boolean isBalanced(String s) {
  Deque<Character> stack = new ArrayDeque<>();
  for (char c : s.toCharArray()) {
    if (c == '(' || c == '[' || c == '{') {
      stack.push(c);
    } else {
      if (stack.isEmpty()) return false;
      char open = stack.pop();
      if (!matches(open, c)) return false;
    }
  }
  return stack.isEmpty();
}
```

A helper method says whether an opener and a closer are of the same type:

```
boolean matches(char open, char close) {
  return (open == '(' && close == ')')
      || (open == '[' && close == ']')
      || (open == '{' && close == '}');
}
```

`isBalanced` pushes each opener. For a closer it returns `false` if the stack is empty; otherwise it pops the latest opener into `open` and returns `false` if `matches(open, c)` fails. Finally, `return stack.isEmpty();` reports whether any bracket was left unclosed.

`matches` joins the three valid pairs with `||` (or); `&&` (and) requires both halves of a pair. For `matches('[', ')')` no pair holds: `false`.

Each character costs *O(1)*: *O(n)* time. At worst (all openers) the stack holds *n* items: *O(n)* memory.

> **Common mistake**
>
> Skipping the empty-stack check for a closer. Without `if (stack.isEmpty()) return false;`, input like `"())"` or `"]"` pops an empty `ArrayDeque`, which throws `NoSuchElementException`. Another typical mistake is returning `true` after the loop, which accepts unclosed strings such as `"(()"`.

7.3.7 Code trace: a valid string

We trace `isBalanced("{{[]}()}[]")`. The string has ten characters, at indices 0 to 9.

Index	`c`	Action	Stack (after)
0	`{`	push	{
1	`{`	push	{{
2	`[`	push	{{[
3	`]`	pop `[`; `matches('[', ']')` true	{{
4	`}`	pop `{`; `matches('{', '}')` true	{
5	`(`	push	{(
6	`)`	pop `(`; `matches('(', ')')` true	{
7	`}`	pop `{`; `matches('{', '}')` true	empty
8	`[`	push	[
9	`]`	pop `[`; `matches('[', ']')` true	empty
(end)	—	`return stack.isEmpty()`	empty → `true`

Table 7.11: Trace of `isBalanced("{{[]}()}[]")`. The stack is written bottom to top.

Whenever a closer arrives, the nearest open opener is on top: the `}` at index 4 matches the `{` at index 1, the one at index 7 matches index 0, found by LIFO alone with no searching. The stack is briefly empty after index 7, so that prefix is valid on its own. At the end it is empty, and the method returns `true`.

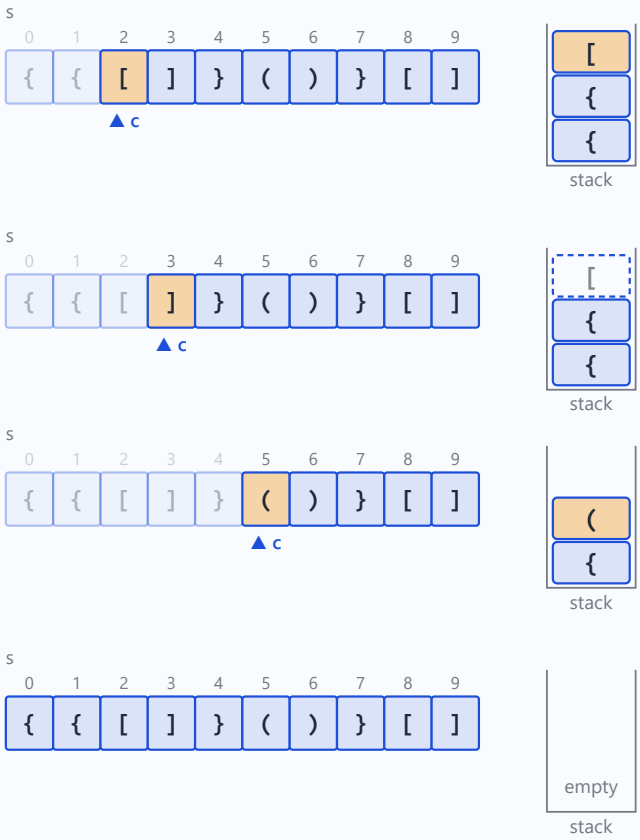


Figure 7.12: Four moments from the trace of `isBalanced("{{[]}()[]")`. (a) Index 2: `[` pushed; the stack is { { [. (b) Index 3: `]` arrives and the latest opener, `[`, comes off. (c) Index 5: `(` pushed; the stack is { (. (d) End of string, stack empty: `true`.

7.3.8 Invalid strings

The algorithm returns `false` in three places, each for a different error.

Example 7.4

Wrong order: `"([)]"`.

1. `(` is pushed; stack: (.
2. `[` is pushed; stack: ([.
3. `)` arrives and the pop yields the latest opener, `[`. `matches('[', ')')` is false, so the method returns `false` without reading the final `]`.

Unclosed: `"(()"`. Two `(` are pushed and `)` removes one. The loop ends with a `(` on the stack; `stack.isEmpty()` is false, so `false` is returned.

Extra closer: `"())"`. `(` is pushed and the first `)` removes it, emptying the stack. When the second `)` arrives, the empty check returns `false`.

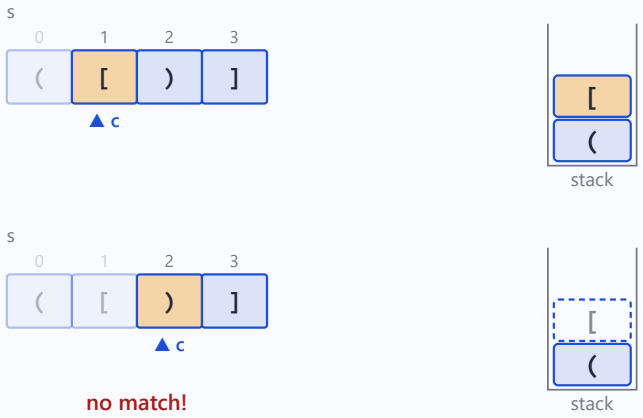


Figure 7.13: `isBalanced("([)]")`. (a) Two openers pushed: the stack is ([. (b) `)` arrives and the pop yields `[`; the types do not match, and the method returns `false`.

Returning early is a bonus: once an error is found, the rest need not be read. The worst case stays *O(n)*, but invalid input often finishes much sooner.

The same algorithm checks that HTML tags such as `<b><i>…</i></b>` nest properly: push each opening tag, pop on each closing tag, and compare names.

7.3.9 Next greater element

Given an array of numbers, find for each item the *first* item to its right that is larger, or −1 if there is none. This is the **next greater element** problem.

index	0	1	2	3	4	5
arr	4	7	3	4	8	1
result	7	8	4	8	−1	−1

Table 7.12: Next greater elements for `arr = {4, 7, 3, 4, 8, 1}`.

By hand: right of 4, the first larger item is 7. Right of 7, 3 and 4 are smaller, so the answer is 8. The 4 next to 3 is larger. Right of 8 there is only 1: −1. The last item has nothing to its right: −1.

Brute force scans right from each item to the first larger one. On a decreasing array every scan runs to the end: $(n-1) + (n-2) + \ldots + 1 = n(n-1)/2$ comparisons, $O(n^2)$ time.

A stack does it in *O(n)*. Scan right to left, keeping only “candidates” on the stack: values that could still answer for items further left. When `x` arrives, pop every value on top that is at most `x`. These lie right of `x` and are no larger; any item left of `x` meets `x` first, and if `x` is not big enough, neither are they, so they can never answer again. The first value not popped, if any, is the answer for `x`, which is then pushed as a candidate.

```
int[] nextGreater(int[] arr) {
    int[] result = new int[arr.length];
    Deque<Integer> stack = new ArrayDeque<>();
    for (int i = arr.length-1; i >= 0; i--) {
        while (!stack.isEmpty() &&
               stack.peek() <= arr[i])
            stack.pop();
        result[i] = stack.isEmpty()
                 ? -1 : stack.peek();
        stack.push(arr[i]);
    }
    return result;
}
```

The stack therefore stays **decreasing** bottom to top, since each item is pushed only after everything at most its size is gone. A stack whose contents stay sorted is a **monotonic stack**.

Theorem 7.2 (Linear time of the monotonic stack)

On an array of n items, `nextGreater` runs in $O(n)$ time and uses $O(n)$ extra memory.

Proof. The loops are nested, but the time is not $O(n^2)$. Each turn of the inner `while` pops once. Every item is pushed exactly once (last line of the outer loop) and never returns once removed, so it is popped at most once: at most n pops over the whole run. The outer loop runs n times with constant work besides the inner loop. The total is on the order of $n + n = 2n$, so $O(n)$. The stack holds at most n items and `result` has n cells: $O(n)$ memory. □

This is amortized analysis again: one outer turn may run the inner loop many times, but the total is bounded. Nested loops do not automatically mean quadratic time; count the inner loop's total runs.

7.3.10 Code trace: nextGreater()

`i`	`arr[i]`	popped	`result[i]`	Stack (after push)
5	1	—	−1 (stack empty)	1
4	8	1 (1 ≤ 8)	−1 (stack empty)	8
3	4	—	8	8 4
2	3	—	4	8 4 3
1	7	3, 4 (≤ 7)	8	8 7
0	4	—	7	8 7 4

Table 7.13: Trace of `nextGreater({4, 7, 3, 4, 8, 1})`. The stack is written bottom to top.

Initially `result` is all zeros (Java zero-fills a new `int` array) and the stack is empty. At `i = 5` the answer is −1, and 1 becomes a candidate. At `i = 4` the 1 on top is popped: it cannot answer for anything left of 8, since 8 is larger and nearer. The stack empties; 8's answer is −1.

Row `i = 1` shows the method best. With 8, 4, 3 on the stack, the incoming 7 pops 3 and 4 and stops at 8, since `8 <= 7` is false; the answer is 8, and 3 and 4 are useless to anything left of 7. At `i = 0` the 7 on top beats 4 at once. The method returns `[7, 8, 4, 8, -1, -1]` after six pushes and three pops, as Theorem 7.2 allows.

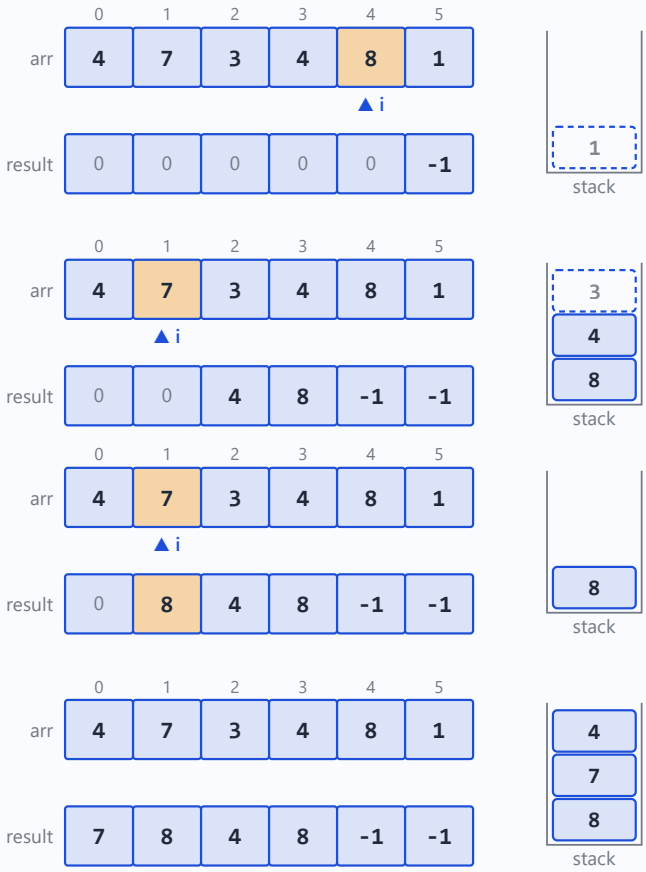


Figure 7.14: Four moments from the `nextGreater` trace. (a) `i = 4`: 1 ≤ 8 on top, popped. (b) `i = 1`: 3 ≤ 7 on top, popped. (c) `i = 1`: 4 is gone too, leaving 8 on top; `result[1] = 8`. (d) The result array is complete: [7, 8, 4, 8, −1, −1].

Common mistake

Writing the inner comparison as `<`. With `stack.peek() < arr[i]`, equal values stay, and an item's "next greater" may be equal to it: in `{5, 5}` the first 5 should get −1, but gets 5. Equal is "not greater," so use `<=`.

7.3.11 A stack that returns its minimum in O(1)

Now add `getMin()`, returning the smallest value on the stack. Scanning takes *O(n)*; we want *O(1)*, like the other operations.

The solution is a second stack. `s` holds all items; the top of `min` is always the smallest value in `s`. A value no larger than the current minimum also goes onto `min`, and a value leaving `s` that equals the top of `min` leaves `min` too, restoring the previous minimum.

```
Deque<Integer> s = new ArrayDeque<>();
Deque<Integer> min = new ArrayDeque<>();
void push(int x) {
  s.push(x);
  if (min.isEmpty() || x <= min.peek())
    min.push(x);
}
int pop() {
  int x = s.pop();
  if (x == min.peek()) min.pop();
  return x;
}
int getMin() { return min.peek(); }
```

Operation	Condition	`s`	`min`	Returns
`push(5)`	`min` empty → push	5	5	—
`push(3)`	3 <= 5 true → push	5 3	5 3	—
`push(7)`	7 <= 3 false	5 3 7	5 3	—
`push(3)`	3 <= 3 true → push	5 3 7 3	5 3 3	—
`getMin()`	—	5 3 7 3	5 3 3	3
`pop()`	3 == 3 → also pop `min`	5 3 7	5 3	3; `getMin()` = 3
`pop()`	7 == 3 false	5 3	5 3	7; `getMin()` = 3
`pop()`	3 == 3 → also pop `min`	5	5	3; `getMin()` = 5

Table 7.14: `push(5)`, `push(3)`, `push(7)`, `push(3)`, `getMin()`, then three `pop()` calls, each followed by `getMin()`. Stacks are written bottom to top.

Each operation does at most two pushes, two pops, or a peek, all *O(1)*; `getMin()` is just `min.peek()`. The price is memory: with pushes in decreasing order, `min` also holds *n* items, *O(n)* extra.

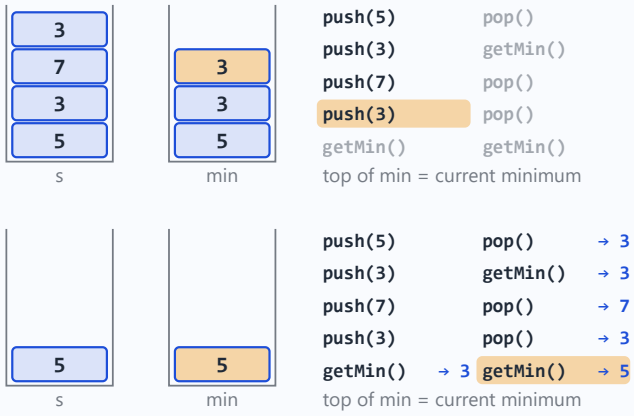


Figure 7.15: The min stack. (a) Second `push(3)`: since 3 <= 3, it also goes onto `min`, which is now 5, 3, 3 bottom to top. (b) After three `pop` calls only 5 remains in `s`; `getMin()` = 5.

Common mistake

Writing the `push` condition as `x < min.peek()`. Equal minima then enter `min` once: above, the second 3 stays out, so the first `pop` also removes the only 3 in `min`, and although `s` still holds a 3, `getMin()` returns 5. With `<=`, each equal minimum gets its own copy.

The `x == min.peek()` test in `pop` is subtle too. `x` is an `int` and `min.peek()` an `Integer`, so Java unboxes and compares values, correctly. Were both sides `Integer`, `==` would compare object identity and could fail for values above 127; then use `equals`.

7.4 Towers of Hanoi and stock span

Two more uses of a stack. In the Towers of Hanoi it makes recursion's hidden call stack explicit; in stock span it keeps "candidates," as in next greater element, turning a quadratic solution into a linear one.

7.4.1 Towers of Hanoi without recursion

The puzzle has three pegs (A, B, C) and n disks of different sizes, stacked on A largest at the bottom, to be moved in the same order to C. Each move takes one top disk, and a larger disk may never go on a smaller one. Each peg is a stack: only its top disk is accessible.

The classic solution is recursive. To move n disks from A to C using B:

1. move the top $n - 1$ disks from A to B (using C);
2. move the largest disk from A to C;
3. move the $n - 1$ disks from B to C (using A).

A single disk (the base case) moves directly. Recursively, the call stack holds the pending subtasks: while step 1 runs, steps 2 and 3 wait. An explicit stack can store each pending job as a *task* instead.

```
// Task: move n disks from peg 'from' to peg 'to' via 'via'
record Task(int n, char from, char via, char to) {}

void hanoi(int n) {
  Deque<Task> stack = new ArrayDeque<>();
  stack.push(new Task(n, 'A', 'B', 'C'));
  while (!stack.isEmpty()) {
    Task t = stack.pop();
    if (t.n() == 1)
      System.out.println(t.from() + " → " + t.to());
    else {                          // in reverse order
      stack.push(new Task(t.n()-1, t.via(), t.from(), t.to()));
      stack.push(new Task(1, t.from(), t.via(), t.to()));
      stack.push(new Task(t.n()-1, t.from(), t.to(), t.via()));
    }
  }
}
```

A `record` (Java 16) is a compact class: `Task` is an immutable object with four fields, read through methods such as `t.n()` and `t.from()`: the disk count `n` and the source, auxiliary, and destination pegs `from`, `via`, `to`. `(3, A, B, C)` means "move 3 disks from A to C using B."

The method pushes the initial task and pops tasks until the stack is empty. A one-disk task is done at once by printing the move. Any other is split into the three subtasks above, pushed in *reverse order*: step 3 `(n-1, via, from, to)`, step 2 `(1, from, via, to)`, then step 1 `(n-1, from, to, via)`, which thus lands on top and comes off next.

The reverse order is LIFO at work. In natural order, step 3 would run first, moving disks from B to C that have not yet left A.

7.4.2 Code trace: hanoi(3)

Table 7.15 shows every iteration of `hanoi(3)`. The stack column lists tasks `(n,from,via,to)` *top to bottom*, so the first comes off next.

Iteration	pop	Action	Stack (after the iteration, top to bottom)
0	—	initial task	(3,A,B,C)
1	(3,A,B,C)	push 3 subtasks	(2,A,C,B) (1,A,B,C) (2,B,A,C)
2	(2,A,C,B)	push 3 subtasks	(1,A,B,C) (1,A,C,B) (1,C,A,B) (1,A,B,C) (2,B,A,C)
3	(1,A,B,C)	move 1: A → C	(1,A,C,B) (1,C,A,B) (1,A,B,C) (2,B,A,C)
4	(1,A,C,B)	move 2: A → B	(1,C,A,B) (1,A,B,C) (2,B,A,C)
5	(1,C,A,B)	move 3: C → B	(1,A,B,C) (2,B,A,C)

Iteration	pop	Action	Stack (after the iteration, top to bottom)
6	(1,A,B,C)	move 4: A → C	(2,B,A,C)
7	(2,B,A,C)	push 3 subtasks	(1,B,C,A) (1,B,A,C) (1,A,B,C)
8	(1,B,C,A)	move 5: B → A	(1,B,A,C) (1,A,B,C)
9	(1,B,A,C)	move 6: B → C	(1,A,B,C)
10	(1,A,B,C)	move 7: A → C	empty

Table 7.15: Trace of `hanoi(3)`. The stack is written top to bottom.

In iteration 1, with `t.n()-1 = 2`, `t.via() = B`, `t.from() = A`, `t.to() = C`, the pushes are `(2,B,A,C)`, `(1,A,B,C)`, `(2,A,C,B)`, leaving "move 2 disks from A to B" on top. Iteration 2 splits it into three one-disk tasks, and iterations 3–5 move the two smallest disks to B (A → C, A → B, C → B). Iteration 6 moves the largest disk from A to C, step 2 of the first split. Iteration 7 splits the remaining "move 2 disks from B to C," and 8–10 finish it.

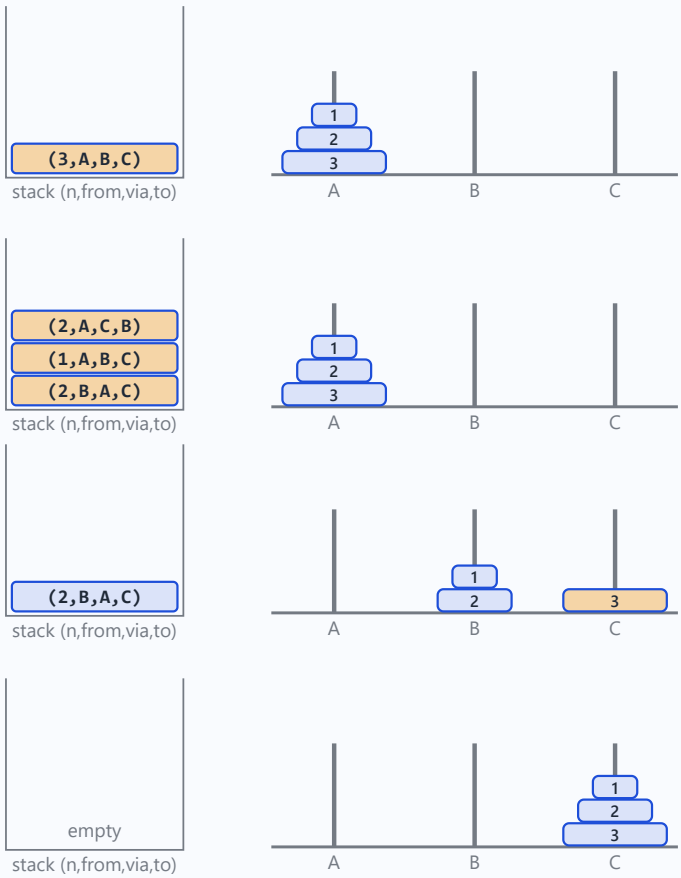


Figure 7.16: The task stack and the pegs. (a) Start: only (3,A,B,C) on the stack; all disks on A. (b) After the first split, top to bottom: (2,A,C,B), (1,A,B,C), (2,B,A,C). (c) Move 4: the largest disk has gone from A to C; the two small disks are on B. (d) The stack is empty and the loop has ended; all three disks are on C.

There are $7 = 2^3 - 1$ moves; in general n disks need $2^n - 1$, since $T(n) = 2T(n - 1) + 1$ and $T(1) = 1$. Slightly more tasks are popped: one per move plus the split ones, so for $n = 3$, 7 + 3 = 10 iterations. The time is $O(2^n)$, a lower bound for the problem itself, since every move must be printed.

Memory is far better: the trace never had more than 5 tasks. Each split replaces one task with three, growing the stack by two, and at most $n - 1$ splits can stack up (each lowers the disk count). So the stack holds at most $1 + 2(n - 1) = 2n - 1$ tasks, $O(n)$: 5 for $n = 3$, 7 for $n = 4$.

An explicit stack puts pending work under the program's control. The runtime limits the call stack, so deep recursion throws `StackOverflowError`, while an explicit stack grows on the heap. The same transformation works for other recursive algorithms, such as depth-first search.

Common mistake

Pushing subtasks in the order they should run. A stack pops the last item first, so the first task must be pushed last: when converting recursion to an explicit stack, push subcalls in reverse order.

7.4.3 The stock span problem

Given a stock's daily closing prices, financial analysts often ask how many days today's price has gone unexceeded: the **stock span** problem.

Definition 7.4

The **span** of a day is the number of consecutive days, counting back from and including that day, whose price is less than or equal to that day's price. The count stops at the first day with a higher price.

Note three details: every span is at least 1, since today counts; equal prices count; and days must be *consecutive*, so the count ends at the first higher price even if lower ones lie beyond.

Example 7.5

Let the prices over 7 days be `[100, 80, 60, 70, 60, 75, 85]`, with days numbered 1 to 7.

1. **Day 1 (100):** no earlier day. Span 1.
2. **Day 2 (80):** the day before has 100 > 80; stop. Span 1.
3. **Day 3 (60):** the day before has 80 > 60. Span 1.
4. **Day 4 (70):** day 3's 60 ≤ 70 counts; day 2's 80 > 70 stops. Span 2.
5. **Day 5 (60):** day 4's 70 > 60. Span 1.
6. **Day 6 (75):** 60, 70, 60 ≤ 75 count; day 2's 80 > 75 stops. Span 4.

7. **Day 7 (85):** 75, 60, 70, 60, 80 ≤ 85 count; day 1's 100 > 85 stops. Span 6.

Result: `[1, 1, 1, 2, 1, 4, 6]`.

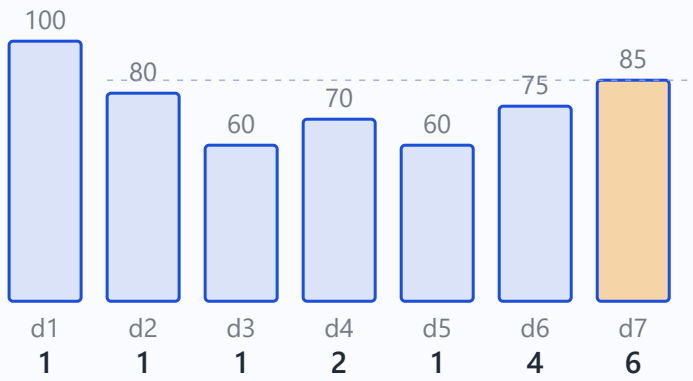


Figure 7.17: Price bars for seven days (d1–d7) with their spans in bold below. For day 7 (85), the five days before it (d2–d6) and the day itself are counted; the dashed line runs over them. The count stops at d1 (100), which is higher than 85.

Brute force walks back from each day to a higher price. With steadily rising prices every walk reaches the start: $1 + 2 + \ldots + n$ steps, $O(n^2)$. A stack removes the repeated walking.

7.4.4 Code: span()

```
int[] span(int[] prices) {
    int[] spans = new int[prices.length];
    Deque<Integer> stack = new ArrayDeque<>();
    stack.push(0); spans[0] = 1;
    for (int i = 1; i < prices.length; i++) {
        while (!stack.isEmpty() &&
            prices[i] >= prices[stack.peek()])
            stack.pop();
        spans[i] = stack.isEmpty() ? i + 1
                                   : i - stack.peek();
        stack.push(i);
    }
    return spans;
}
```

The idea is next greater element's, but scanning left to right with *indices* on the stack: those of days not yet exceeded, with prices decreasing bottom to top. When day `i` arrives, days priced at or below today are popped. They can never limit a future day: today has exceeded them and is a nearer, higher (or equal) barrier.

The top is then the latest earlier day with a *higher* price, and the span is the distance to it, `i - stack.peek()`. If the stack is empty, today beat every earlier price and the span is `i + 1` (days 0 to `i`). Finally today is pushed as a possible future barrier.

The first day, with span 1, is handled before the loop by `stack.push(0); spans[0] = 1;`, so the loop starts at `i = 1`. Since the stack holds indices, the condition `prices[i] >= prices[stack.peek()]` reaches prices through them.

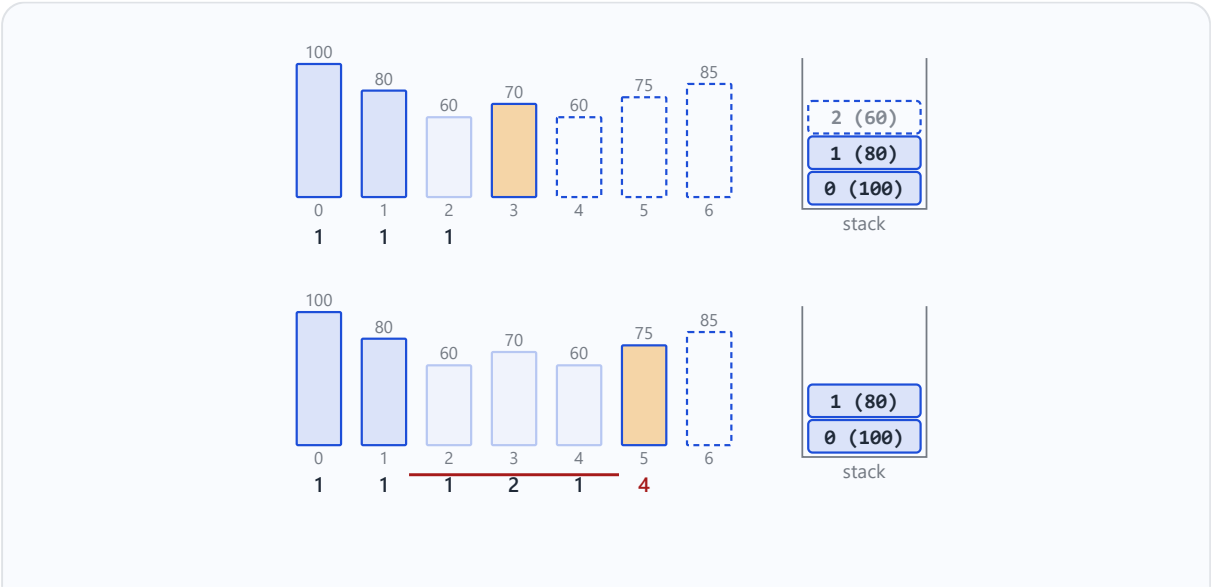
The cost argument is that of next greater element: each index is pushed once and popped at most once, so time and memory are *O(n)*.

7.4.5 Code trace: span()

`i`	Price	popped	Top	`spans[i]`	Stack (after push)
0	100	—	—	1 (first day)	0(100)
1	80	—	0	1 − 0 = 1	0(100) 1(80)
2	60	—	1	2 − 1 = 1	0(100) 1(80) 2(60)
3	70	2 (60 ≤ 70)	1	3 − 1 = 2	0(100) 1(80) 3(70)
4	60	—	3	4 − 3 = 1	0(100) 1(80) 3(70) 4(60)
5	75	4, 3	1	5 − 1 = 4	0(100) 1(80) 5(75)
6	85	5, 1	0	6 − 0 = 6	0(100) 6(85)

Table 7.16: Trace of `span({100, 80, 60, 70, 60, 75, 85})`. The stack lists indices bottom to top, each with that day's price in parentheses.

At `i = 3`, 70 beats day 2 (60), which is popped: day 3 (70) is nearer and higher for any later day. Day 1 (80) is on top, so the span is *3 − 1 = 2*. At `i = 5`, 75 pops days 4 (60) and 3 (70) and stops at day 1 (80): span *5 − 1 = 4*. At `i = 6`, 85 pops days 5 (75) and 1 (80) and stops at day 0 (100): span 6. The result `[1, 1, 1, 2, 1, 4, 6]` matches Example 7.5, after 7 pushes and 5 pops.



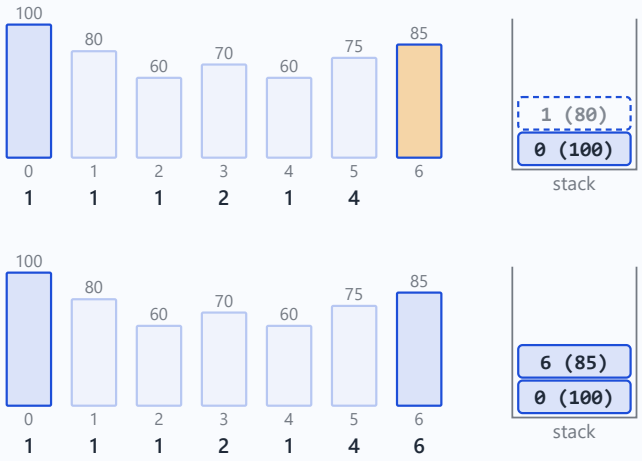


Figure 7.18: Four moments from the `span` trace. (a) `i = 3`: 70 ≥ 60, day 2 is popped. (b) `i = 5`: days 3 and 4 are gone, day 1 (80) is on top; span = 4. (c) `i = 6`: 85 ≥ 80, day 1 is popped too. (d) End: the stack holds 0(100) and 6(85); the spans are [1, 1, 1, 2, 1, 4, 6].

Common mistake

Taking the span as `i` when the stack empties; days are numbered from 0, so day `i` ends a run of `i + 1` days. Another typical mistake is using `>` in the inner loop, which leaves equal-priced days uncounted, though the definition says "less than or equal."

7.5 Expression notations

An expression like `3 + 4 * 2` is easy for people but hard for a computer in two ways: the multiplication is done before the addition written to its left, and parentheses, possibly nested, change the order. This section writes expressions in two forms that make the order explicit, and converts between them with a stack.

7.5.1 Infix, prefix, and postfix

An expression has **operands** (A, B, 5) and **operators** (+, −, *, /). The three notations differ in where the operator goes.

Notation	Operator position	A * (B + C) / D
Infix	between the operands	`A * ( B + C ) / D`
Prefix · Polish notation	before the operands	`/ * A + B C D`
Postfix · reverse Polish notation	after the operands	`A B C + * D /`

Table 7.17: Three notations for the same expression.

Everyday notation is **infix**, as in `B + C`. In **prefix** the operator comes first, `+ B C`; this is *Polish notation*, after the logician Jan Łukasiewicz, who proposed it in the 1920s. In **postfix** it comes last, `B C +`: *reverse Polish notation*, or RPN.

To read postfix, scan `A B C + * D /` from the left; at each operator, the two values just before it are its operands and are replaced by the result. `B C +` → `(B + C)`; then `A (B + C) *` → `A * (B + C)`; then `… D /` → `(A * (B + C)) / D`.

Prefix and postfix **need neither parentheses nor precedence rules**: the order of operations follows from the sequence of symbols. Postfix is evaluated left to right, prefix right to left, in one pass with a stack, which is why compilers and interpreters usually convert to postfix. Java bytecode is stack-based: `a + b` becomes "push a, push b, add." Some calculators (older HP models) work in RPN, as do symbolic algebra programs and expression-tree algorithms.

> **Common mistake**
>
> Moving operators forward in their infix order to get prefix. For `A * (B + C) / D`, `* A / + B C D` is wrong: it means `A * ((B + C) / D)`, which can differ under integer division and is a different computation tree anyway. Infix applies `*` and `/` left to right, `A * (B + C)` first, so the correct prefix is `/ * A + B C D`, with the outermost operation first.

7.5.2 Precedence, associativity, and converting by hand

Infix needs two rules: **precedence** says which of two different operators goes first, and **associativity** in which direction equal-precedence operators apply.

Operator	Precedence	Associativity
`^` (power)	3	right to left
`*` `/`	2	left to right
`+` `-`	1	left to right

Table 7.18: Operator precedence and associativity.

Left associativity makes `A - B + C` mean `(A - B) + C`: `10 - 4 + 3` is `6 + 3 = 9`, whereas grouping from the right gives `10 - 7 = 3`. Its postfix is `A B - C +`. Exponentiation is right-associative: `2 ^ 3 ^ 2` is `2 ^ (3 ^ 2) = 2^9 = 512`, not `(2 ^ 3) ^ 2 = 64`.

To convert by hand, use **full parenthesization**:

1. Following precedence and associativity, put every operation in its own parentheses.
2. For **postfix**, move each operator to the right end of its parentheses and delete the parentheses.
3. For **prefix**, move each operator to the left end of its parentheses and delete the parentheses.

Example 7.6

(a) `A * (B + C) / D`. The parentheses first, then `*` and `/` left to right: `((A * (B + C)) / D)`.

- Postfix: `(B + C)` → `B C +`; `(A * …)` → `A B C + *`; `(… / D)` → `A B C + * D /`.
- Prefix: `(B + C)` → `+ B C`; `(A * …)` → `* A + B C`; `(… / D)` → `/ * A + B C D`.

(b) `A + B * C - D / E`. `*` and `/` first, then `+` and `-` left to right: `((A + (B * C)) - (D / E))`.

- Postfix: left side `A B C * +`, right side `D E /`, and `-` last: `A B C * + D E / -`.
- Prefix: `-` first, then the left side `+ A * B C`, then the right side `/ D E`: `- + A * B C / D E`.

A quick check: **operands keep the same order in all three notations**; only operators move. A conversion that scrambles the operands is certainly wrong. The converse fails: operands can be in order with operators misplaced (like `* A / + B C D` in 7.5.1).

7.5.3 Infix to postfix: the algorithm

A program would have to parse the whole expression to parenthesize it. Edsger Dijkstra's 1961 *shunting-yard* algorithm, named after a railway switching yard, instead makes one left-to-right pass: operators and opening parentheses wait on a stack, operands go straight to the output.

Symbol read	Action
Operand (A, B, 5 …)	Write it to the output.
`(`	Push it.
`)`	Pop and output until `(` appears; discard the `(`.
Operator `op`	While the top is not `(` and its precedence is ≥ that of `op`, pop and output; then push `op`.
End of input	Pop and output all remaining operators.

Table 7.19: Rules for infix-to-postfix conversion.

The reasoning: a newly read operator still lacks its right operand, so it waits on the stack. When `op` arrives, waiting operators of precedence at least `op`'s have complete operands and must be applied first, so they are output; lower ones apply after `op` and keep waiting.

Popping on equality (≥) reflects left associativity: in `A - B + C`, when `+` arrives, the equal `-` must go first and is output, giving `A B - C +`. For right-associative `^`, only strictly greater precedence (>) pops, so in `A ^ B ^ C` the second `^` goes above the first and applies first.

A parenthesis walls off precedence: operators pushed after `(` never look beneath it, and `)` pops only down to it. Parentheses are never output; postfix needs none.

Each symbol is read once and each operator pushed and popped once: *O(n)* time for an expression of length *n*, and *O(n)* stack memory at worst.

7.5.4 Code: infixToPostfix()

```
String infixToPostfix(String expr) {
  StringBuilder out = new StringBuilder();
  Deque<Character> stack = new ArrayDeque<>();
  for (char c : expr.toCharArray()) {
    if (Character.isLetterOrDigit(c)) out.append(c);
    else if (c == '(') stack.push(c);
    else if (c == ')') {
      while (stack.peek() != '(')
        out.append(stack.pop());
      stack.pop();                  // discard '('
    } else {                        // operator
      while (!stack.isEmpty() &&
          prec(stack.peek()) >= prec(c))
        out.append(stack.pop());
      stack.push(c);
    }
  }
  while (!stack.isEmpty()) out.append(stack.pop());
  return out.toString();
}
```

The helper that gives the precedences:

```
int prec(char op) {
  switch (op) {
    case '+': case '-': return 1;
    case '*': case '/': return 2;
    default: return 0;   // '('
  }
}
```

The code applies the five rules of Table 7.19 in order. A letter or digit (`Character.isLetterOrDigit(c)`) is an operand and is appended to `out` , a `StringBuilder` , Java's efficient way to build a string piece by piece. A `(` is pushed. On `)` , `while (stack.peek() != '(')` outputs the operators down to the `(` , and one more `stack.pop()` discards it. Anything else is an operator: the `while` loop pops while the top's precedence is not lower than the incoming one's, then pushes the newcomer. At the end the remaining operators are appended.

The condition "the top is not `(` " is handled by `prec` : its `default` branch gives `(` precedence 0, below every real operator, so `prec('(') >= prec(c)` is never true. Only `)` pops a `(` .

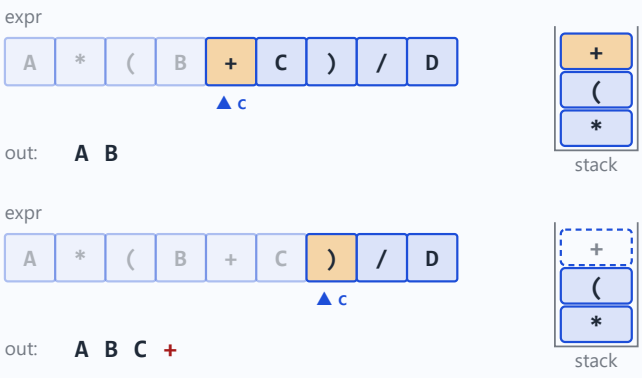
The code assumes single-character operands, only `+ - * /` (no `^`), no spaces, and a valid expression. Given a `)` with no matching `(`, `stack.peek()` returns `null` on the empty stack, and `null != '('` throws `NullPointerException` while unboxing the `Character`.

7.5.5 Code trace: A*(B+C)/D

`c`	Action	Stack (after)	`out`
`A`	operand → output	empty	A
`*`	stack empty → push	*	A
`(`	always push	* (	A
`B`	operand → output	* (	A B
`+`	`(` on top (precedence 0) → push	* (+	A B
`C`	operand → output	* (+	A B C
`)`	pop to `(` : `+` to output; `(` discarded	*	A B C +
`/`	top `*` (2) ≥ `/` (2) → `*` to output; stack empty → push `/`	/	A B C + *
`D`	operand → output	/	A B C + * D
(end)	remaining `/` to output	empty	A B C + * D /

Table 7.20: Trace of `infixToPostfix("A*(B+C)/D")`. The stack is written bottom to top.

The decisive moment is reading `/`: the ≥ rule outputs the equal-precedence `*`, which is on the left in infix and goes first. Had it stayed, the result would be `A B C + D / *`, meaning `A * ((B + C) / D)`. The result matches the hand conversion of Example 7.6.



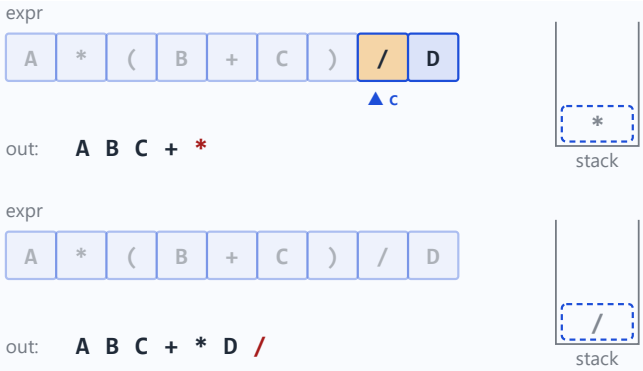


Figure 7.19: Four moments from converting `A*(B+C)/D`. (a) `+` read: with `(` on top, it is pushed directly. (b) `)` read: `+` goes to the output. (c) `/` read: the equal-precedence `*` goes to the output. (d) End of input: the remaining `/` is output; the result is A B C + * D /.

7.5.6 Code trace: A+B*C-D/E

With no parentheses, precedence does all the work.

`c`	**Action**	**Stack (after)**	`out`
`A`	to output	empty	A
`+`	stack empty → push	+	A
`B`	to output	+	A B
`*`	top `+` (1) < 2 → push	+ *	A B
`C`	to output	+ *	A B C
`-`	`*` (2) ≥ 1 → output; `+` (1) ≥ 1 → output; push `-`	−	A B C * +
`D`	to output	−	A B C * + D
`/`	top `-` (1) < 2 → push	− /	A B C * + D
`E`	to output	− /	A B C * + D E
(end)	`/`, then `-` to output	empty	A B C * + D E / −

Table 7.21: Trace of `infixToPostfix("A+B*C-D/E")`.

At `*`, the lower-precedence `+` keeps waiting and `*` goes on top: the stack's version of "multiplication first." At `-`, two operators leave: the higher `*`, then the equal `+`. At the end the stack empties from the top, `/` then `-`. The result `A B C * + D E / -` means `(A + B·C) − D/E`.

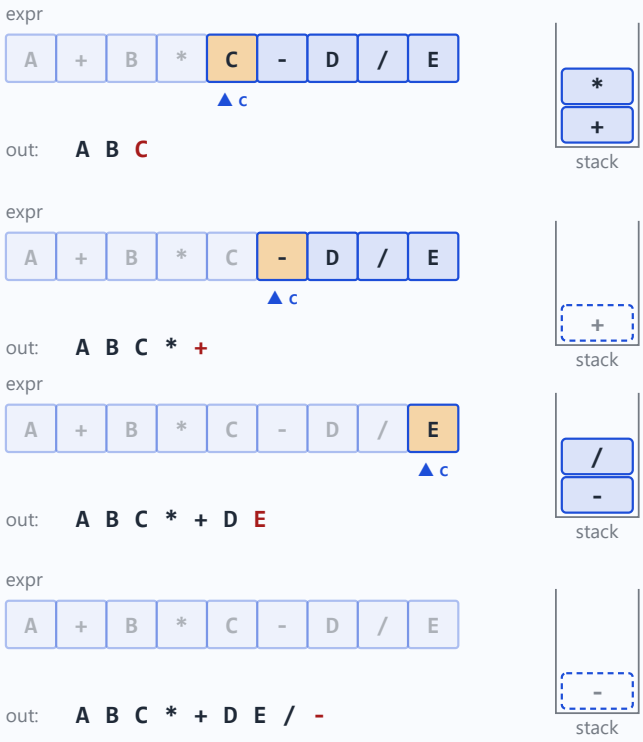


Figure 7.20: Four moments from converting `A+B*C-D/E`. (a) `*` sits on the lower-precedence `+`; output A B C. (b) `-` arrives: `*` has been output, and now the equal-precedence `+` leaves. (c) `/` is on `-`, and E has been output. (d) End: the stack is emptied; the result is A B C * + D E / −.

Common mistake

Popping only *one* operator when an operator arrives. The rule is a `while` loop. In `A+B*C-D/E`, popping only `*` at `-` and leaving `+` would give `A B C * D E / - +`, meaning `A + (B·C − D/E)`, which breaks left associativity.

7.5.7 Evaluating a postfix expression

The payoff of postfix is trivial evaluation. Read left to right:

1. Push each number.
2. On an operator, pop two items, **the right operand first, then the left**; apply the operator and push the result.
3. At the end, one item remains on the stack: the result.

No parentheses or precedence are involved; each symbol is handled once: *O(n)*.

A helper does the arithmetic:

```
int apply(int left, char op, int right) {
  switch (op) {
    case '+': return left + right;
    case '-': return left - right;
    case '*': return left * right;
    default:  return left / right;
  }
}
```

The evaluation method:

```
int evaluatePostfix(String expr) {
  Deque<Integer> stack = new ArrayDeque<>();
  for (char c : expr.toCharArray()) {
    if (Character.isDigit(c)) {
      stack.push(c - '0');
    } else {
      int right = stack.pop();
      int left = stack.pop();
      stack.push(apply(left, c, right));
    }
  }
  return stack.pop();
}
```

`c - '0'` turns a digit character into its value: characters are numeric codes, and `'0'` through `'9'` are consecutive, so `'7' - '0'` is 7. On an operator, the first `pop` goes into `right`, the second into `left`, and `apply(left, c, right)` is pushed. A final `pop` returns the result. The code assumes single-digit numbers, no spaces, and a valid expression; `/` is integer division, so `7 / 2` is 3.

Why right first? The left operand was read, and pushed, earlier, so it sits lower. For addition and multiplication this is irrelevant, but not for subtraction and division: in `5 2 -`, the first `pop` gives 2 (`right`), the second 5 (`left`), and the result is *5 − 2 = 3*, not *2 − 5 = −3*.

> **Common mistake**
>
> Taking the first `pop` as the left operand. With `int left = stack.pop(); int right = stack.pop();`, every subtraction and division comes out backward. The first `pop` is the right operand, the second the left.

7.5.8 Code trace: evaluatePostfix()

Substituting `A = 4`, `B = 1`, `C = 2`, `D = 3` into `A B C + * D /` from Example 7.6 gives the postfix string `"412+*3/"`, whose infix equivalent is *4 · (1 + 2) / 3 = 4*.

`c`	Action	Stack (after)
`4`	digit → push 4	4
`1`	digit → push 1	4 1
`2`	digit → push 2	4 1 2
`+`	`right` = 2, `left` = 1; `apply(1, '+', 2)` = 3 → push	4 3
`*`	`right` = 3, `left` = 4; `apply(4, '*', 3)` = 12 → push	12
`3`	digit → push 3	12 3
`/`	`right` = 3, `left` = 12; `apply(12, '/', 3)` = 4 → push	4
(end)	`return stack.pop()`	empty → result 4

Table 7.22: Trace of `evaluatePostfix("412+*3/")`. The stack is written bottom to top.

The stack holds intermediate results not yet used by an operator; at `+`, 1 and 2 are replaced by 3. The stack sizes (1, 2, 3, 2, 1, 2, 1) also reveal validity: each number adds one, each binary operator removes one, and in a valid expression the stack holds at least 2 items at every operator and exactly 1 at the end.

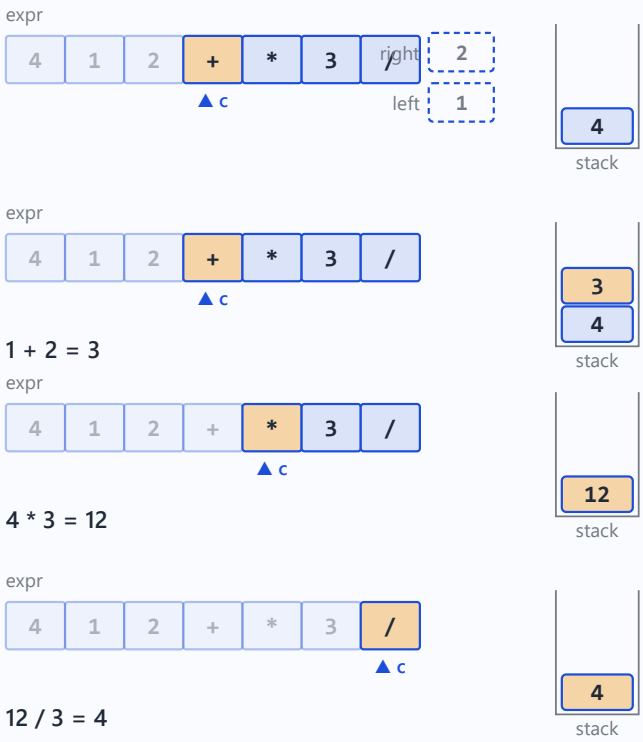


Figure 7.21: Four moments from evaluating `"412+*3/"`. (a) `+` arrives: `right` = 2 comes off, then `left` = 1. (b) 1 + 2 = 3 is pushed. (c) 4 · 3 = 12 is pushed. (d) 12 / 3 = 4; one value remains.

7.5.9 Infix to prefix

Infix to prefix needs no new algorithm, just the postfix one, slightly modified, between two reversals:

1. Reverse the infix expression.
2. Swap the parentheses: `(` ↔ `)`.
3. Apply the postfix algorithm, but **do not pop on equal precedence**: pop only operators of strictly greater precedence (>).
4. Reverse the output.

Step	Result
1. Reverse	`D / ) C + B ( * A`
2. Swap parentheses	`D / ( C + B ) * A`
3. Convert to postfix (no pop on equal)	`D C B + A * /`
4. Reverse → prefix	`/ * A + B C D`

Table 7.23: Converting `A * (B + C) / D` to prefix.

Prefix is the mirror image of postfix. Reversal flips each operator's position relative to its operands, so taking the postfix of the reversed expression and reversing it puts each operator in front. Reversal also flips the parentheses, which step 2 corrects.

The "no pop on equal" rule preserves associativity, which reversal turns from left to right. When `*` arrives in the reversed expression, the equal `/` stays and `*` goes above it, so `*` applies first, and after the final reversal `A * (B + C)` still precedes `/ D`. Popping on equality would give `D C B + / A *` and the prefix `* A / + B C D`, the very mistake of 7.5.1.

For the second example, `A + B * C - D / E`, the same method yields `- + A * B C / D E`, the hand result of Example 7.6.

7.5.10 Code trace: infixToPrefix()

```
String rev(String s) {
  return new StringBuilder(s).reverse().toString();
}
String infixToPrefix(String expr) {
  String t = rev(expr);            // 1. reverse
  t = swapParens(t);               // 2. ( ⇔ )
  String p = postfixForPrefix(t);  // 3. no pop on equal
  return rev(p);                   // 4. reverse
}
// in main:
System.out.println(infixToPrefix("A*(B+C)/D"));
```

Four steps, four lines. `rev` uses `StringBuilder`'s built-in `reverse()`. `swapParens`, whose body the slides omit, swaps `(` and `)`:

```
String swapParens(String s) {
  StringBuilder sb = new StringBuilder();
  for (char c : s.toCharArray()) {
    if (c == '(') sb.append(')');
    else if (c == ')') sb.append('(');
    else sb.append(c);
  }
  return sb.toString();
}
```

`postfixForPrefix` copies `infixToPostfix` except for the operator-branch comparison:

```
for (char c : expr.toCharArray()) {
  if (Character.isLetterOrDigit(c)) out.append(c);
  else if (c == '(') stack.push(c);
  else if (c == ')') {
    while (stack.peek() != '(')
      out.append(stack.pop());
    stack.pop();                  // discard '('
  } else {                        // operator
    while (!stack.isEmpty() &&
        prec(stack.peek()) > prec(c))
      out.append(stack.pop());
    stack.push(c);
  }
}
while (!stack.isEmpty()) out.append(stack.pop());
```

With `>` instead of `>=`, an equal-precedence operator stays. The trace for the reversed, swapped input `"D/(C+B)*A"`:

`c`	Action	Stack (after)	`out`
`D`	to output	empty	D
`/`	stack empty → push	/	D
`(`	push	/ (	D
`C`	to output	/ (	D C
`+`	`(` on top → push	/ (+	D C
`B`	to output	/ (+	D C B
`)`	`+` to output, `(` discarded	/	D C B +
`*`	top `/` (2) > 2? no → push	/ *	D C B +
`A`	to output	/ *	D C B + A
(end)	`*`, then `/` to output	empty	D C B + A * /

Table 7.24: Trace of `postfixForPrefix("D/(C+B)*A")`.

Reversing `p = "DCB+A*/"` gives `"/*A+BCD"`, which `main` prints; the operands are still in order A, B, C, D.

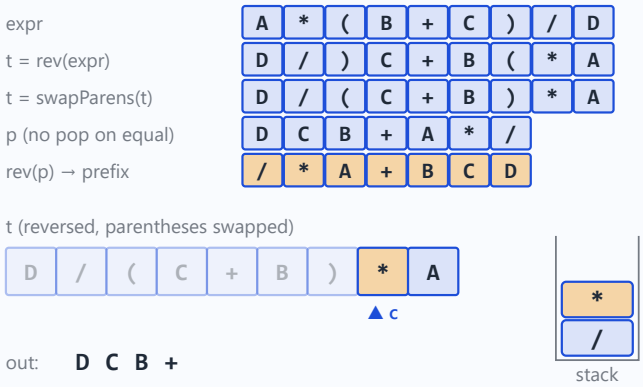


Figure 7.22: Converting to prefix. (a) The four stages, one under another: `expr`, the reversed `t`, `t` with parentheses swapped, `p` found without popping on equality, and the reversed result `/ * A + B C D`. (b) The key moment of step 3: when `*` arrives, the `/` on top has equal precedence and is not popped; `*` goes on top of it.

Each of the four steps (two reversals, one parenthesis swap, one conversion) is *O(n)*, so the whole is *O(n)*.

7.5.11 Evaluating a prefix expression

Prefix evaluation follows the postfix rules with two changes: read **right to left**, and **the first pop is the left operand**, since reading from the right pushes the right operand first and the left one ends up on top.

Example 7.7

Evaluate the prefix expression `- * 3 4 / 8 2` (in infix, *3 · 4 − 8 / 2*). Symbols are read right to left; the stack is written bottom to top.

1. `2` → push. Stack: 2.
2. `8` → push. Stack: 2 8.
3. `/` → `left` = 8 (first pop), `right` = 2; *8 / 2 = 4* → push. Stack: 4.
4. `4` → push. Stack: 4 4.
5. `3` → push. Stack: 4 4 3.
6. `*` → `left` = 3, `right` = 4; *3 · 4 = 12* → push. Stack: 4 12.
7. `-` → `left` = 12, `right` = 4; *12 − 4 = 8* → push. Stack: 8.

The result is 8. Swapping the order in step 7 would give *4 − 12 = −8*.

Postfix and prefix evaluation are one stack idea run in two directions: the stack holds operands awaiting an operator, and only the reading direction, hence the pop order, differs.

7.6 Common mistakes and a common pattern

Most stack bugs fall into a few types. The table gathers the chapter's mistakes into a checklist for reviewing stack code.

Mistake	What happens?	Fix
`pop` / `peek` on an empty stack	An exception or `null`, depending on the library; `"())"` triggers exactly this in the parentheses check.	Check `isEmpty()` first.
Confusing `top` and `size`	Writing outside the array, or a wrong item count.	`top` is the index of the last item; `size = top + 1`; full means `top == capacity - 1`.
Not clearing the vacated cell	The object escapes the garbage collector (memory leak, loitering).	In an object array, `data[top--] = null`.
Operand order in postfix evaluation	Subtraction and division come out backward.	First `pop` is the right operand, second the left: `5 2 -` = 5 − 2 = 3.
Iteration order	Two stacks with the same contents print in opposite orders.	`ArrayDeque` iterates top to bottom, the old `Stack` bottom to top; know which you use.
Infinite recursion	Frames pile up: `StackOverflowError` (the call stack overflows).	Every recursive method needs a reachable base case.

Table 7.25: Common mistakes in stack code.

Iteration order comes from library internals. `java.util.Stack`, a `Vector`, keeps items in insertion order, so `toString()` and `for-each` go bottom to top. `ArrayDeque`'s `push` inserts at the front, where iteration starts, so it goes top to bottom. After pushing 1, 2, 3, `Stack` prints `[1, 2, 3]` and `ArrayDeque` `[3, 2, 1]`: same contents, opposite output.

One pattern runs through the chapter. A stack is the right tool when the question is **"Which item, seen most recently, is still unresolved?"**: the latest bracket opened but not closed, the latest method called but not returned, the latest intermediate result not yet used, the nearest candidate not yet exceeded (next greater element, stock span), the latest subtask not yet done (Hanoi). When a problem raises this question, its solution very likely uses a stack.

7.7 Summary

- A **stack** is a **LIFO** (last-in, first-out) ADT with insertion and removal only at the top. `push`, `pop`, `top` / `peek`, `isEmpty`, and `size` are all *O(1)*; searching is *O(n)*; *n* items take *O(n)* memory.

- Popping an empty stack is **underflow**; pushing onto a full bounded stack is **overflow**. Libraries treat empty stacks differently, so check `isEmpty()` before `pop` and `peek`.
- In an **array-based stack**, `top` is the last item's index (−1 when empty) and `size = top + 1`; vacated object cells are set to `null`. A **dynamic array** doubles when full: a resizing `push` is *O(n)*, but *n* pushes cost under *3n*, so `push` is **amortized *O(1)***. Growing by a fixed amount costs $O(n^2)$ in total.
- In a **linked-list stack** the top is the head, where `push` and `pop` work in *O(1)*. It overflows only when memory runs out, and each node carries an extra reference. New Java code should use `Deque<E> s = new ArrayDeque<>()`.
- A browser's **back/forward** uses two stacks, method calls the **call stack**. Recursion needs memory proportional to its depth; without a base case it ends in `StackOverflowError`.
- **Balanced parentheses** keeps unmatched openers on a stack and catches three kinds of error (wrong order, unclosed, extra closer) in *O(n)*.
- **Next greater element** and **stock span** are solved in *O(n)* with a **monotonic stack**, decreasing bottom to top, because each item is pushed once and popped at most once. A stack with *O(1)* `getMin()` keeps the minima on a second stack (using `<=`).
- The **Towers of Hanoi** yield to an explicit task stack that pushes subtasks in reverse order: $2^n - 1$ moves, *O(n)* stack.
- **Infix** becomes **postfix** in *O(n)* with an operator stack, popping while the top's precedence is ≥ the incoming one's. For **prefix**: reverse, swap parentheses, convert without popping on equality, reverse again. Operands keep their order in all three notations.
- **Postfix evaluation** reads left to right; at an operator the first `pop` is the right operand, the second the left. Prefix is read right to left, and the first pop is the left operand.

Problem	On the stack	Time
`push` / `pop` / `top` / `isEmpty` / `size`	—	O(1)
String reversal	characters	O(n)
Balanced parentheses	unmatched opening brackets	O(n)
Next greater element	candidate values to the right (decreasing)	O(n)
Stock span	indices of days not yet exceeded	O(n)
Towers of Hanoi	subtasks (n, from, via, to)	$O(2^n)$
Infix → postfix / prefix	operators and `(`	O(n)
Postfix evaluation	intermediate results	O(n)
Min stack (`getMin` **)**	the current minimum	O(1)
Back / forward navigation	visited pages (two stacks)	O(1)

Table 7.26: Problems solved with a stack.

7.8 Key terms

Each term links to the course glossary, which gives its Turkish equivalent and its use in other chapters.

stack · top · push · pop · LIFO (last-in, first-out) · abstract data type (ADT) · interface · generic · overflow · underflow · capacity · dynamic array · amortized analysis · linked list · singly linked list · node · reference · undo · call stack · stack frame · recursion · base case · backtracking · balanced parentheses · next greater element · stock span · infix · prefix · postfix · depth-first search (DFS) · time complexity · space complexity

7.9 Test Yourself

1. An empty stack receives `push(7)`, `push(2)`, `pop()`, `push(9)`, `top()`, `push(4)`, `pop()`, `pop()`. What do the calls return, and what is left?

   Answer

   The first `pop()` returns 2, leaving 7. After `push(9)`, `top()` returns 9 without removing it. After `push(4)` the stack is 7 9 4; the pops return 4, then 9. The values are 2, 9, 4, 9, and only 7 remains.

2. What do `isBalanced("([]{})")` and `isBalanced("([]{)}")` return? For the invalid one, at which character is the error detected?

   Answer

   The first returns `true`: `(`, `[` pushed; `]` pops `[`; `{` pushed; `}` pops `{`; `)` pops `(`; the stack ends empty. The second returns `false`: when the `)` at index 4 arrives, the stack is ({ and the pop yields `{`; `matches('{', ')')` is false. The final `}` is never read.

3. What are the postfix and prefix forms of `(A + B) * C - D / (E + F)`?

   Answer

   Fully parenthesized: `(((A + B) * C) - (D / (E + F)))`. Postfix: `A B + C * D E F + / -`. Prefix: `- * + A B C / D + E F`. The operands stay in order A to F in both. In `infixToPostfix`, `-` pops the `*`, while at `/` the lower-precedence `-` stays.

4. What is the value of the postfix expression `7 2 - 3 * 8 4 / +`? What is on the stack after each

operator?

Answer

`-`: right 2, left 7 → 5; stack 5. `*`: right 3, left 5 → 15; stack 15. `/`: right 4, left 8 → 2; stack 15 2. `+`: right 2, left 15 → 17. The value is 17; in infix, *(7 − 2) · 3 + 8 / 4*.

5. What are the spans for prices `[50, 40, 45, 60, 55, 70]`? How many pops does `span` make in total?

Answer

Spans: `[1, 1, 2, 4, 1, 6]`. At `i = 2`, 45 pops day 1 (40) and stops at day 0 (50): 2 − 0 = 2. At `i = 3`, 60 pops days 2 and 0, emptying the stack: 3 + 1 = 4. At `i = 5`, 70 pops days 4 and 3, emptying it: 6. That is 5 pops (indices 1, 2, 0, 4, 3), fewer than the 6 pushes.

6. What is the next greater element array for `{3, 8, 2, 5, 1, 9}`? Which values are popped at step `i = 1`?

Answer

The result is `[8, 9, 5, 9, 9, -1]`. Right to left: 9 → −1; 1 → 9; 5 pops 1 → 9; 2 → 5; at `i = 1` 8 arrives with the stack 9 5 2 bottom to top, pops 2 and then 5, and the 9 left on top is the answer. Finally 3 → 8.

7. An empty stack of capacity 2 that doubles when full receives 12 pushes. How many resizes and copies are made? Is the total cost below *3n*?

Answer

Resizes occur on the 3rd `push` (2 → 4, 2 copies), the 5th (4 → 8, 4 copies), and the 9th (8 → 16, 8 copies): 3 resizes, 14 copies. With 12 writes the total is 26, below *3 · 12 = 36*. The final capacity is 16.

8. Which moves does the stack version of `hanoi(2)` print? Which tasks are on the stack (top to bottom) after the first iteration?

Answer

In the first iteration `(2,A,B,C)` is split, and the stack, top to bottom, becomes `(1,A,C,B)`, `(1,A,B,C)`, `(1,B,A,C)`. The next three iterations carry them out: A → B, A → C, B → C, a total of $2^2 - 1 = 3$ moves.

9. Suppose the min stack's `push` condition reads `x < min.peek()` instead of `x <= min.peek()`.

What happens on `push(2)`, `push(2)`, `pop()`, `getMin()` from an empty stack?

Answer

The first 2 goes onto `min`; the second does not, as `2 < 2` is false. `pop()` removes a 2, and since `2 == min.peek()`, the only 2 in `min` goes too. `s` still holds a 2 but `min` is empty, so `min.peek()` in `getMin()` returns `null`, and unboxing it to `int` throws `NullPointerException`.

10. What is the value of the prefix expression `/ * 4 + 1 2 3`?

Answer

Right to left: 3, 2, 1 are pushed (stack 3 2 1). `+`: left 1 (first pop), right 2 → 3; stack 3 3. Push 4; stack 3 3 4. `*`: left 4, right 3 → 12; stack 3 12. `/`: left 12, right 3 → 4. The value is 4; in infix, *4 · (1 + 2) / 3*.

7.10 Further reading

- M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, *Data Structures and Algorithms in Java* (6th ed., 2014), Section 6.1: the stack ADT, array and linked-list implementations, and bracket matching and HTML tag checking, in Java code close to these notes.
- R. Sedgewick and K. Wayne, *Algorithms* (4th ed., 2011), Section 1.3: stacks implemented with a dynamic array and a linked list, and Dijkstra's two-stack expression evaluation algorithm in a short, clear example.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms* (3rd ed., 2009), Sections 10.1 and 17.4: the array implementation of a stack and the amortized analysis of dynamic tables, in more formal language.

7.11 References

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 3rd ed., MIT Press, 2009.
2. E. W. Dijkstra, "Algol 60 translation: An Algol 60 translator for the X1 and Making a translator for Algol 60", Report MR 35/61, Mathematisch Centrum, Amsterdam, 1961.
3. M. T. Goodrich, R. Tamassia, M. H. Goldwasser, *Data Structures and Algorithms in Java*, 6th ed., Wiley, 2014.
4. R. Lafore, *Data Structures and Algorithms in Java*, 2nd ed., Sams, 2002.
5. Oracle, *Java SE 17 API Specification*: documentation for `java.util.Deque` and `java.util.ArrayDeque`.
6. R. Sedgewick, K. Wayne, *Algorithms*, 4th ed., Addison-Wesley, 2011.

PART II · CHAPTER 8

Queues and Deques

A queue is a waiting line in software: first come, first served. Items enter at one end and leave at the other, and none may cut in. The rule appears everywhere, from processor sharing and print jobs to network buffers and breadth-first search. We start with the queue ADT and Java's `Queue` interface, build a queue on an array, see why the naive version wastes space, and fix it with a circular queue whose indices wrap with mod. Linked queues, kinds of queues, and the deque, which works at both ends, follow, with the sliding window maximum. Applications traced in code close the chapter: round-robin scheduling, the Josephus problem, binary number generation, and a queue from two stacks.

Code for this chapter: `github.com/sercankulcu/data-structures-java/Bolum08/src`

LEARNING OUTCOMES

The FIFO model
Explain first-in, first-out behavior; find a queue's contents and return values after a sequence of operations.

Implementation
Implement a queue with an array and a linked list; wrap circular indices with mod, tell empty from full, and grow a full queue without breaking its order.

Kinds of queues
Tell the circular queue from the deque; implement a deque with a doubly linked list and use it where it fits.

Applications
Pick the right queue for round-robin scheduling, the Josephus problem, binary number generation, sliding window maximum, and a queue from two stacks, and justify each cost.

8.1 The queue ADT

A data structure can be described at two levels: *what* it does (its operations, what each changes and returns) and *how* (how the data sits in memory and what steps each operation takes). A description at the first level is an **abstract data type** (ADT). This section treats the queue at that level, apart from

any implementation.

8.1.1 What is a queue?

Think of a supermarket checkout. Newcomers join at the back, the cashier serves the person at the front, and everyone behind moves up; nobody cuts in or jumps ahead. Ticket windows, bank tellers, and toll booths follow the same rule: first come, first served.

A **queue** imitates this. Its items form a line whose ends have separate jobs: new items join at the **rear**, and removal happens at the **front**. The item that leaves is always the one that has waited longest.

> **Definition 8.1**
>
> A **queue** is an abstract data type that keeps items in a sequence, with insertion only at the rear and removal only at the front. This discipline is called **FIFO** (first-in, first-out).

A stack inserts and removes at the same end, so the last item in is the first out (LIFO); a queue uses opposite ends, so the first in is the first out. Push 1, 2, 3 into each and remove three times: the stack returns 3, 2, 1, the queue 1, 2, 3. A queue exists to hold work *in order of arrival* and hand it back in that order.

Each end of a queue is used in one direction only. The queue is a **linear** structure: every item but the first has one predecessor, every item but the last one successor, and nothing branches.

> **Common mistake**
>
> Mixing up the two ends. New items join at the *rear*, and the item removed is the *oldest*, not the newest. Thinking "the last one added comes out first" means using the queue as a stack.

8.1.2 The basic operations

A queue has two basic operations, whose names are used throughout the course:

- **Insertion** (`enqueue`) adds an item at the rear — joining the line.
- **Removal** (`dequeue`) removes and returns the front item, the first one in.

In a good implementation both run in *O(1)* time, however many items the queue holds. That requires reaching both ends directly, without searching, and the whole design of the array and linked versions serves this goal.

Items in the middle cannot be reached. This restriction is deliberate: client code cannot break the order or let an item skip ahead, and the implementation only has to make two ends fast.

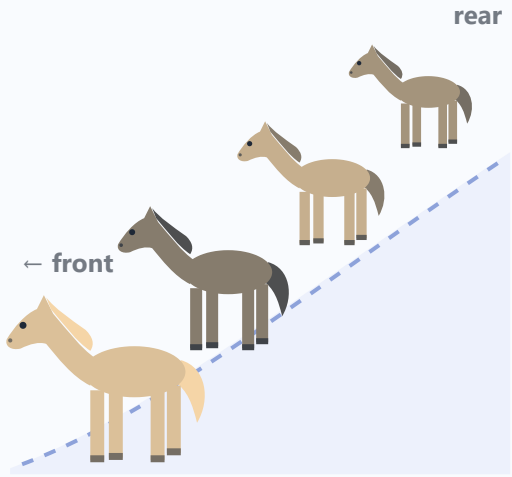


Figure 8.1: Horses walking single file make a good picture of a queue. The horse in front goes first, a newcomer joins at the back, and no horse can pass the one ahead.

8.1.3 Operations and interface of the queue ADT

A few helper operations complete the queue ADT (Table 8.1).

Operation	Description
`enqueue(e)`	Adds `e` at the rear.
`dequeue()`	Removes and returns the front item.
`first()` / `peek()`	Returns the front item without removing it.
`rear()`	Returns the rear item without removing it.
`size()`	Returns the number of items.
`isEmpty()`	Is the queue empty?
`isFull()`	Is the queue full? (Fixed-capacity implementations only.)

Table 8.1: Operations of the queue ADT.

Only `enqueue` and `dequeue` change the queue; the rest just report. `first()` differs from `dequeue()` as `top()` differs from `pop()` on a stack: both return the front item, but `first()` leaves it in place, which is what you want when you must look before deciding whether to remove.

`isFull()` makes sense only for a fixed capacity. A linked queue, or one that grows when full, is never "full" while memory lasts.

In Java an ADT is naturally written as an **interface**, which gives method signatures without bodies. Here is a shortened form of the queue interface from the textbook by Goodrich, Tamassia, and Goldwasser:

```
public interface Queue<E> {
    int size();
    boolean isEmpty();
    void enqueue(E e);
    E first();    // null if empty
    E dequeue();  // null if empty
}
```

`E` is a type parameter fixed when the queue is created: `Queue<Integer>` holds integers, `Queue<String>` strings. Classes and interfaces with type parameters are **generic**. The interface omits `rear()`, which basic use doesn't need, and deliberately omits `isFull()` so that it also covers unbounded queues.

The comments record a contract: on an empty queue, `first()` and `dequeue()` return `null`. Removing from an empty queue is **underflow**; adding to a full fixed-capacity queue is **overflow**. How to report these errors is a design choice. Goodrich's interface returns `null`, other implementations throw; in these notes the circular queue and the linked queue throw and the deque returns `null`. The caller feels the difference: an unchecked `null` lets the error travel silently, while an exception stops the program at once.

Common mistake

Calling `dequeue()` without checking for an empty queue. If the implementation returns `null`, using the result directly (`int x = q.dequeue();`) tries to unbox `null` into an `int` and throws `NullPointerException`. Ask `isEmpty()` first.

8.1.4 Queue operations step by step

Apply `enqueue(20)`, `enqueue(10)`, `enqueue(15)`, and three `dequeue()` calls to an empty queue whose end references are named `front` and `rear`.

Example 8.1

1. **Start.** The queue is empty; `front` and `rear` are both `null`.
2. `enqueue(20)`: As the only item, 20 is both front and rear; both references point to it. Size 1.
3. `enqueue(10)`: 10 joins at the rear; `rear` moves to it, `front` stays at 20. Size 2.
4. `enqueue(15)`: The queue is 20, 10, 15 from front to rear. Size 3.
5. `dequeue()`: 20 is removed and returned; `front` moves to 10. Size 2.
6. `dequeue()`: 10 leaves; `front` and `rear` both point to 15. Size 1.
7. `dequeue()`: 15 leaves; the queue is empty and both references are `null` again.

The items came out as 20, 10, 15, the order they went in — the FIFO rule.

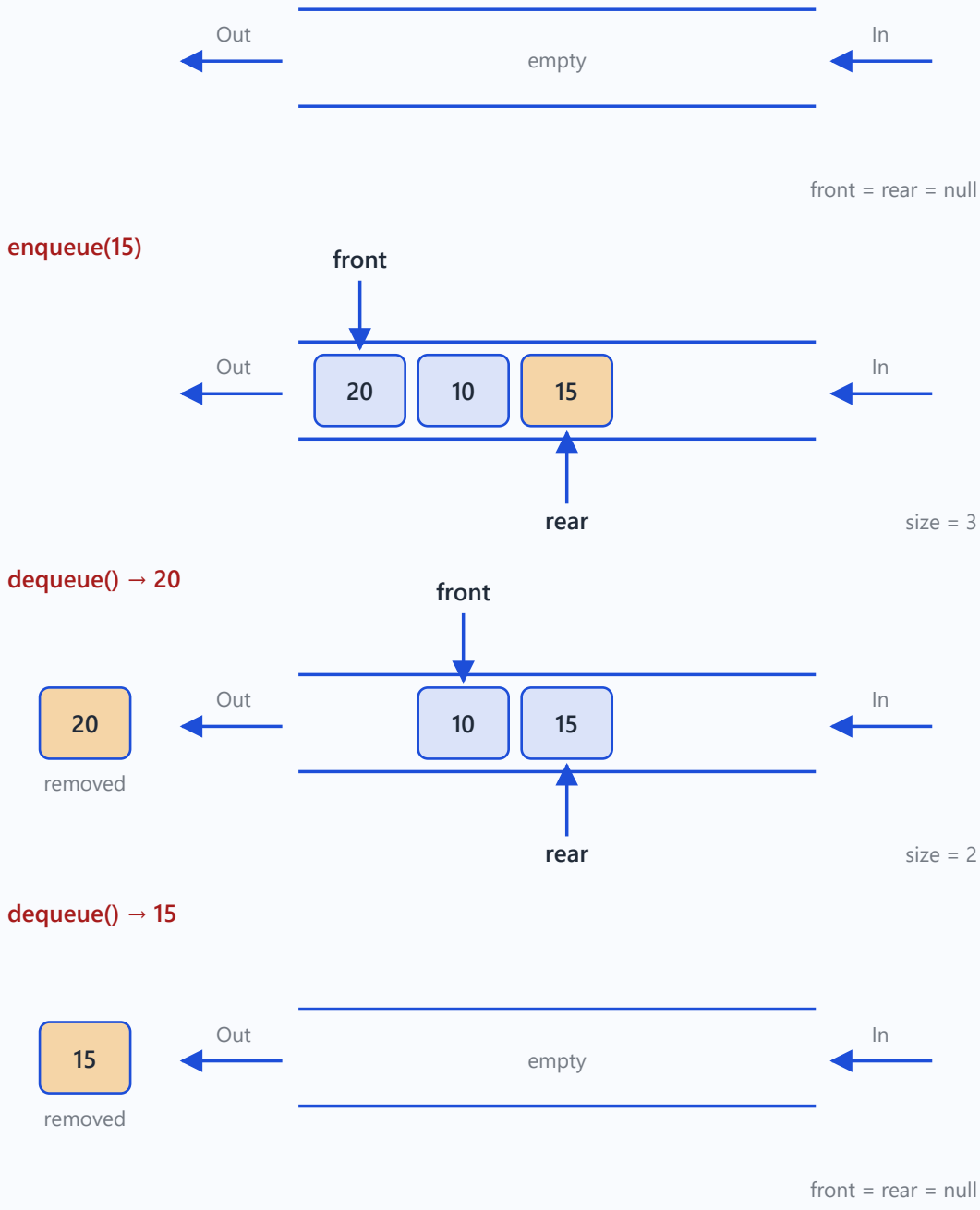


Figure 8.2: Four moments of Example 8.1, top to bottom. (a) Empty: `front = rear = null`; items enter at "In" on the right and leave at "Out" on the left. (b) After three `enqueue` calls: 20, 10, 15, with `front` on 20 and `rear` on 15. (c) The first `dequeue()` has returned 20; `front` is on 10. (d) The last item has left, and both references are `null`.

Two edge cases matter. The first insertion into an empty queue must update both references, since that item is both front and rear; removing the last item must reset `rear` to `null` along with `front`. Most queue bugs come from these two cases.

A tip for exercises: after each operation, write the contents on one line *from front to rear* with the returned value beside them. Writing always in one direction keeps you from removing at the wrong end.

8.1.5 Queues in Java: java.util.Queue

Java provides the `java.util.Queue<E>` interface, in which each operation comes in two forms: one throws an exception on failure, the other returns a special value (Table 8.2).

Operation	Throws exception	Returns special value
Insert	`add(e)`	`offer(e)` → `false`
Remove	`remove()`	`poll()` → `null`
Examine	`element()`	`peek()` → `null`

Table 8.2: Methods of `java.util.Queue`.

On an empty queue, `remove()` and `element()` throw `NoSuchElementException`, while `poll()` and `peek()` return `null`. When a bounded queue is full, `add(e)` throws `IllegalStateException` and `offer(e)` returns `false`; on unbounded queues, such as the two classes below, `offer` always returns `true`.

An interface has no objects of its own; two common classes implement it:

- `ArrayDeque` uses a circular array and is usually the fastest; it is a growable version of the circular queue in these notes.
- `LinkedList` uses a linked list; it also supports list operations but creates a node object per item.

The code below runs the basic operations on an `ArrayDeque`. Declaring the variable as `Queue<Integer>` limits the code to queue operations and makes the implementing class easy to swap.

```
Queue<Integer> q = new ArrayDeque<>();
q.offer(20);
q.offer(10);
q.offer(15);        // [20, 10, 15]
q.peek();           // 20, not removed
q.poll();           // 20 → [10, 15]
q.poll();           // 10 → [15]
q.size();           // 1
q.poll();           // 15 → []
q.poll();           // null (empty)
```

Call	Returns	Queue (after)
`offer(20)`	`true`	[20]
`offer(10)`	`true`	[20, 10]
`offer(15)`	`true`	[20, 10, 15]
`peek()`	20	[20, 10, 15]
`poll()`	20	[10, 15]
`poll()`	10	[15]
`size()`	1	[15]
`poll()`	15	[]
`poll()`	`null`	[]

Table 8.3: Trace of the `Queue` example, front to rear.

`peek()` returns 20 without changing the queue, so the next `poll()` returns 20 again. The last `poll()`, on an empty queue, returns `null`; `remove()` there would have thrown `NoSuchElementException`.

Our own classes use the ADT names `enqueue` / `dequeue`; the Java library does the same work with `offer` / `poll` / `peek`. The names differ, the behavior does not.

> **Common mistake**
>
> Expecting `peek()` to throw on an empty queue. It belongs to the special-value group and returns `null`; its throwing twin is `element()`. Rule of thumb: `add`, `remove`, `element` throw; `offer`, `poll`, `peek` return `false` or `null`.

8.1.6 Where queues are used

A queue fits wherever work should be done in arrival order. Four common cases:

- **Operating system schedulers.** Ready processes wait in a *ready queue* and get the CPU in turn. Round-robin scheduling, the simplest form of this idea, is covered in Section 8.5.1.
- **Print queues.** A printer prints the documents sent to it in order of arrival; later ones wait.
- **Buffers.** Keyboard input, network packets, and video streams wait in a buffer, which evens out the speed difference between producer and consumer: data piles up while the producer is ahead and drains while the consumer catches up.
- **Breadth-first search** (BFS). Nodes discovered in a graph or tree are queued, so nearby nodes are visited before distant ones and the traversal goes level by level. Binary number generation in Section 8.5.4 is a small instance.

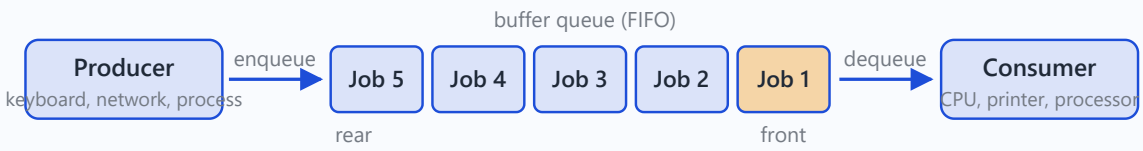


Figure 8.3: The producer–consumer pattern. The producer (keyboard, network, process) adds jobs at the rear (`enqueue`); the consumer (processor, printer) takes the front job (`dequeue`). The buffer queue absorbs their speed difference. Job 1 is the oldest and goes first.

What these cases share is fairness and preserved order. A printer must print the first document first; packets handled out of order make a video show scrambled frames; a scheduler must not starve a process. A queue guarantees all this by structure: once an item is in, its turn comes as soon as the items ahead of it have left.

8.2 Array-based queues

The obvious way to implement a queue is with an array, whose cells are reached by index in *O(1)*; the queue only has to track its two ends. We try the simplest idea first, see why it wastes space, and fix it with a circular queue.

8.2.1 The array representation

The items live in an array, and two indices mark the ends:

- `front` : the index of the first item, which the next removal takes.
- `rear` : the index of the last item, the most recently added.

To insert, increment `rear` and write to `data[rear]` ; to remove, take `data[front]` and increment `front` . Items never move, so both operations are *O(1)*. The method is simple and fast, but the capacity is fixed.

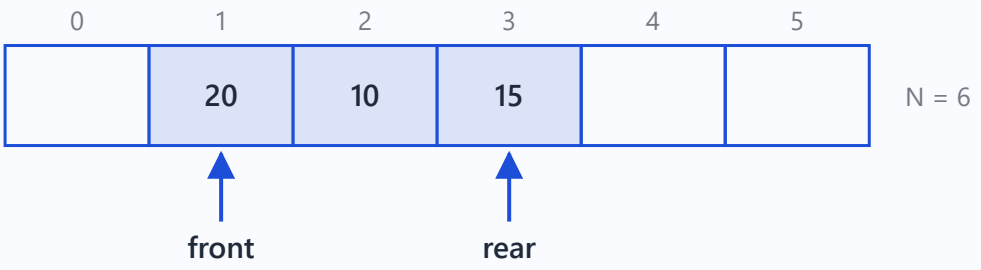


Figure 8.4: A queue in an array of capacity $N = 6$: 20, 10, 15 at indices 1–3, with `front = 1` and `rear = 3` . The item at index 0 was removed earlier; its slot is free, but the simple array queue cannot reuse it.

Figure 8.4 already shows the weak spot. Removals push the front rightward, and the slots freed behind it are never used again. `rear` creeps toward the end of the array according to the number of insertions so far, not the number of items held.

8.2.2 The trouble with a simple array queue

Take a simple array queue of capacity $N = 6$ with `front = 0` and `rear = -1` (nothing added yet; the first insertion raises `rear` to 0 and writes at index 0). We insert 10, 20, 30, 40, remove three times, then try to insert 50, 60, 70.

Operation	front	rear	Array	Returns
(start)	0	−1	[_, _, _, _, _, _]	—
`enqueue(10)`	0	0	[10, _, _, _, _, _]	—
`enqueue(20)`	0	1	[10, 20, _, _, _, _]	—
`enqueue(30)`	0	2	[10, 20, 30, _, _, _]	—
`enqueue(40)`	0	3	[10, 20, 30, 40, _, _]	—
`dequeue()`	1	3	[_, 20, 30, 40, _, _]	10
`dequeue()`	2	3	[_, _, 30, 40, _, _]	20
`dequeue()`	3	3	[_, _, _, 40, _, _]	30
`enqueue(50)`	3	4	[_, _, _, 40, 50, _]	—
`enqueue(60)`	3	5	[_, _, _, 40, 50, 60]	—
`enqueue(70)`	3	5	[_, _, _, 40, 50, 60]	"full"

Table 8.4: Trace of a simple array queue ($N = 6$). The array is written from index 0 to 5; "_" marks a free slot.

Four insertions raise `rear` to 3. Three removals return 10, 20, 30 and move `front` to 3, so `front` and `rear` both point at 40, the only item left. Inserting 50 and 60 brings `rear` to 5, the last index ($N - 1$). For `enqueue(70)`, `rear + 1 = 6 = N`: there is no room at the end, and the queue reports "full."

Yet it holds only 40, 50, and 60, while indices 0–2 sit empty: half the array is wasted because freed slots cannot be reused. Worse, even if each insertion were followed at once by a removal, `rear` would hit the end after six insertions. A simple array queue's lifetime is bounded by its total insertions, not by its contents.

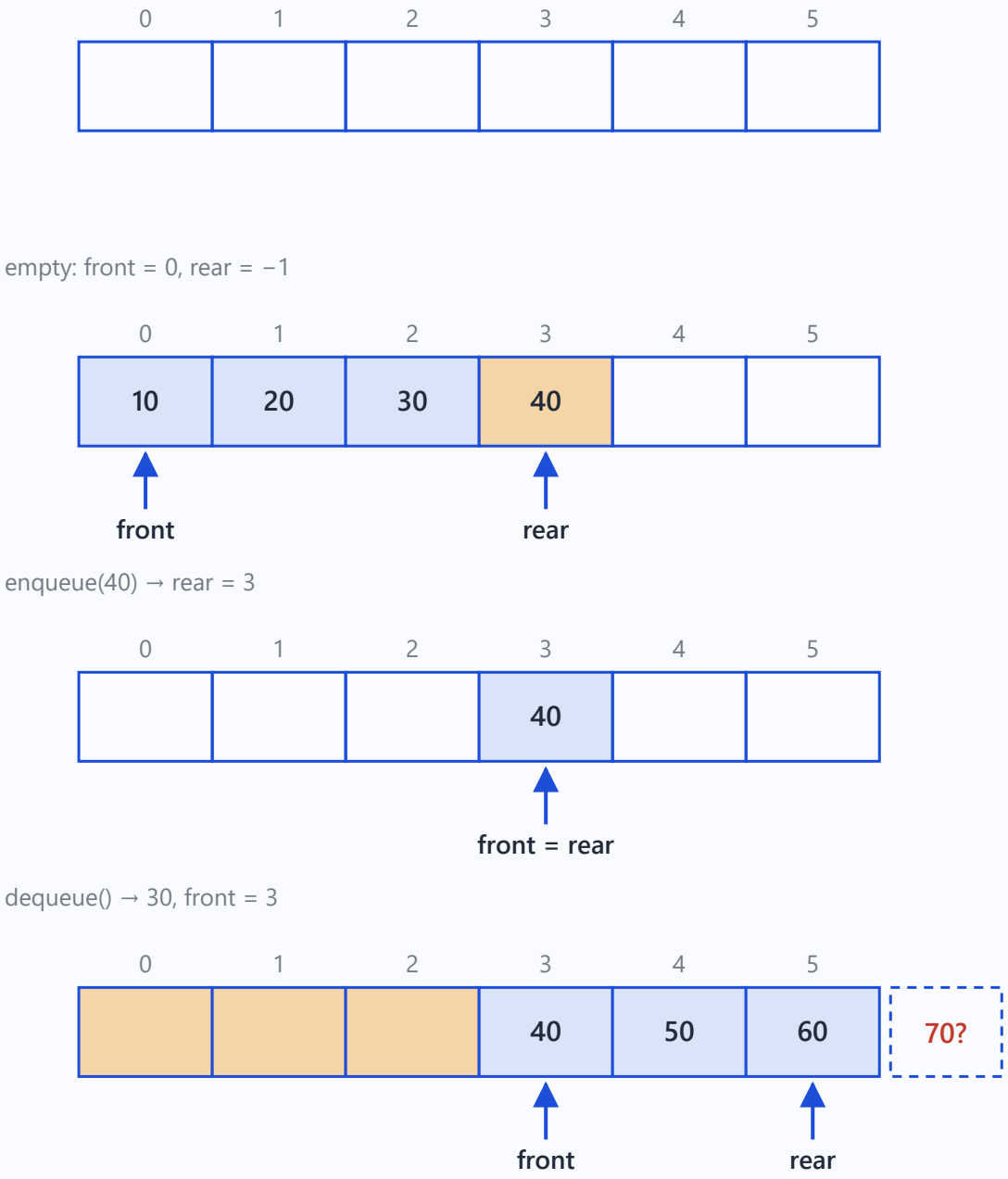


Figure 8.5: Four moments of Table 8.4. (a) Empty: `front = 0`, `rear = -1`. (b) After four insertions, `rear = 3`. (c) After three removals, `front = rear = 3`; the freed slots are lost. (d) After 50 and 60, `rear = 5 = N − 1`, and `enqueue(70)` reports "full" although the three orange slots are free.

There are two ways out.

Solution 1: shifting. After each removal, shift the remaining items one slot left. The front then stays at `data[0]` and needs no index; `size` counts the items:

```
public int dequeue() {              // simple array queue with shifting
    if (size == 0)
        throw new IllegalStateException("empty");
    int result = data[0];           // the front is always data[0]
    for (int i = 1; i < size; i++)
        data[i - 1] = data[i];      // everyone moves one slot left
    size--;
    return result;
}
```

With *n* items, a removal makes *n − 1* assignments and costs *O(n)*; emptying a queue of 1,000 items takes *999 + 998 + … + 1 + 0 = 499,500*. The waste is gone, and so is the speed.

Solution 2: a circular array. Indices wrap from the end of the array to the start. Nothing is shifted, freed slots are reused, and every operation stays *O(1)*. This is the circular queue.

> **Common mistake**
>
> Treating `rear == N - 1` as "the queue is full." It only means the end of the array has been reached. In the last row of Table 8.4 the queue holds 3 items with 3 slots free, yet no insertion is possible.

8.2.3 The circular queue

In a simple array queue the end of the array is a wall. A circular queue removes it: after the last slot comes the first, so the two ends join logically into a ring. In memory the array is still a flat row; the ring exists only in how indices are computed.

> **Definition 8.2**
>
> A **circular queue** stores its items in an array of *N* slots and advances its indices with mod, as in *(i + 1) % N*, so index *N − 1* is followed by 0. It is also called a **ring buffer**.

A circular queue is still FIFO; only its "next index" changes, from `i + 1` to `(i + 1) % N`. The two agree while *i < N − 1*; at *i = N − 1*, *(N − 1 + 1) % N = N % N = 0*, and the index wraps. Freed slots are reused and nothing is shifted.

Our circular queue keeps two fields: `front`, the index of the first item, and `size`, the item count. The rear is not stored; the free slot for the next item is `(front + size) % N`.

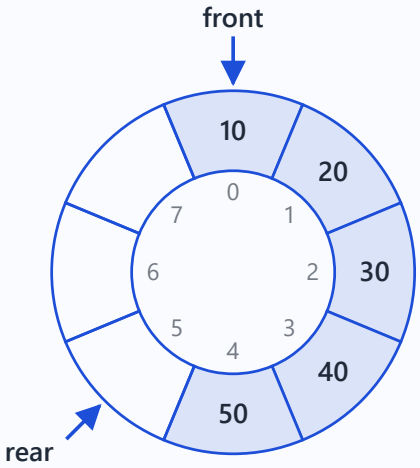


Figure 8.6: Ring view of a circular queue of capacity 8. Indices 0–4 hold 10–50 and `front` is 0. Here `rear` is the free slot for the next item: *(front + size) % N = (0 + 5) % 8 = 5*. Slot 7 is followed by slot 0.

The ring has no "end": slot 7 neighbors both 6 and 0. Insertion advances the rear clockwise and removal moves the front clockwise, so the queue can start and end anywhere on the ring.

8.2.4 Computing indices with mod

Every index is computed from `front`, `size`, and *N*. For non-negative *a*, the remainder *a % N* lies between 0 and *N − 1*, always a valid index. Table 8.5 evaluates the four basic formulas for *N = 8*, `front = 6`, `size = 4`.

What?	Formula	Example
Next free slot (insertion point)	`(front + size) % N`	(6 + 4) % 8 = 2
Last item	`(front + size − 1) % N`	(6 + 3) % 8 = 1
Item k from the front (k = 0, 1, ...)	`(front + k) % N`	k = 2 → (6 + 2) % 8 = 0
`front` **after a removal**	`(front + 1) % N`	(6 + 1) % 8 = 7

Table 8.5: Index formulas for a circular queue (example: *N = 8*, `front = 6`, `size = 4`).

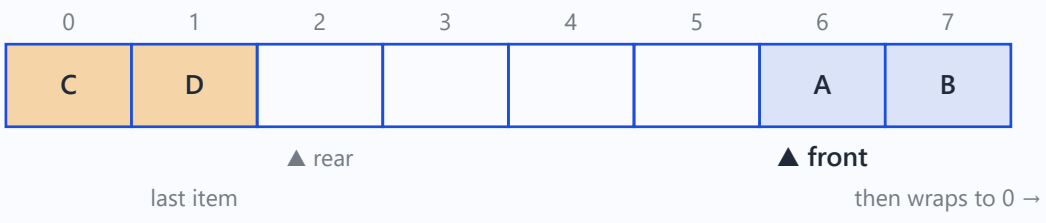


Figure 8.7: Array view of Table 8.5: A (6), B (7), C (0), D (1) from front to rear. `front = 6`, the last item is at index 1, and the next free slot is 2. After index 7 comes 0.

All the formulas rest on one idea: item *k* lies *k* steps past the front. In a plain array that is *front + k*; in a circular one the sum may run past the end, and `% N` wraps it around. C is item 2: *6 + 2 = 8* is outside the array, but *8 % 8 = 0* is C's real place.

The next free slot is simply the case `k = size`: positions 0 to `size - 1` are occupied, so the last item is at `size - 1`. Mod handles the jump from end to start in one expression, with no `if` asking whether we have reached the end.

Since *front < N* and *size ≤ N*, these sums stay below *2N*, so `%` wraps at most once. Stepping *backward* needs care: in Java the result of `%` takes the sign of the dividend, so for `front = 0`, `(front - 1) % N` gives *−1 % N = −1*. The correct expression is `(front - 1 + N) % N`, which is exactly what inserting at the front of an array-based deque uses.

Common mistake

Forgetting the mod. Written as `front + size`, the insertion point runs past *N* once the queue has wrapped, and `ArrayIndexOutOfBoundsException` follows; in the example *6 + 4 = 10* is no index of an 8-slot array.

8.2.5 Full or empty?

Two indices look sufficient at first: `front` for the first item and `rear` for the next write position. But then empty and full look alike. In an empty queue both point to the same slot; after *N* insertions `rear` has circled the ring back to `front`. Either way, `front == rear`.

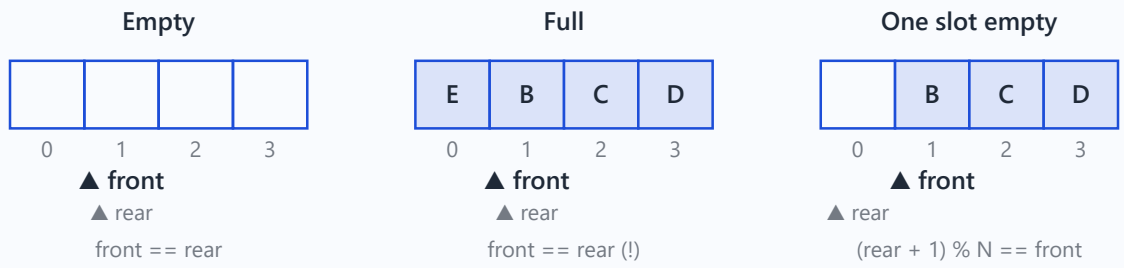


Figure 8.8: Three situations for *N = 4*. Left, empty: `front == rear`. Middle, full (B, C, D, E from front to rear): again `front == rear`. Right, the "one empty slot" method: with B, C, D stored, slot 0 stays empty and the full test is *(rear + 1) % N == front*.

Two common remedies exist.

1. Keep a counter. A `size` field makes the tests `size == 0` (empty) and `size == N` (full), and all *N* slots are used. These notes and the slides use this method; with the counter, the rear index need not be stored, since it is `(front + size) % N`.

2. Leave one slot empty. The queue holds at most $N - 1$ items. It is empty when `front == rear` and full when `(rear + 1) % N == front`, with `rear` just behind `front` and one empty slot between. No counter is needed, but a slot is always wasted.

Example 8.2

A circular queue with $N = 7$ using the "one empty slot" method starts at `front = rear = 3`. How many items fit, and what is `rear` when full?

1. **Start:** `front == rear == 3`; the queue is empty.
2. **Insertions:** `rear` advances 3 → 4 → 5 → 6 → 0 → 1 → 2, wrapping from 6 to 0 on the fourth insertion.
3. **Full check:** After six insertions `rear = 2` and *(2 + 1) % 7 = 3 = front*: full, so a seventh insertion is refused.

The queue stores $N - 1 = 6$ items and ends with `rear = 2`. A seventh item would move `rear` onto `front`, making full indistinguishable from empty.

The methods differ little. The counter costs one integer field but uses every slot, and `size == 0` and `size == N` say plainly what is tested. Leaving a slot empty pays off where a shared counter is costly or risky: when a producer and a consumer use one buffer concurrently, the producer changes only `rear` and the consumer only `front`, so neither must update a common field.

Common mistake

Keeping only `front` and `rear` while storing all N items. Then `front == rear` holds for both empty and full, and the code misreads one of them: it either overwrites the oldest item of a full queue or treats a full queue as empty. Use a counter or an empty slot.

8.2.6 A circular queue step by step

Here is a counter-based circular queue with $N = 5$, starting at `front = 0`, `size = 0`. `enqueue` writes to slot `(front + size) % 5` and increments `size`; `dequeue` returns `data[front]`, clears the slot, sets `front` to `(front + 1) % 5`, and decrements `size`.

Operation	Slot written / read	`front`	`size`	Array	Returns
(start)	—	0	0	[␣, ␣, ␣, ␣, ␣]	—
`enqueue(10)`	(0 + 0) % 5 = 0	0	1	[10, ␣, ␣, ␣, ␣]	—
`enqueue(20)`	(0 + 1) % 5 = 1	0	2	[10, 20, ␣, ␣, ␣]	—
`enqueue(30)`	(0 + 2) % 5 = 2	0	3	[10, 20, 30, ␣, ␣]	—
`dequeue()`	0	1	2	[␣, 20, 30, ␣, ␣]	10

Operation	Slot written / read	`front`	`size`	Array	Returns
`dequeue()`	1	2	1	[_, _, 30, _, _]	20
`enqueue(40)`	(2 + 1) % 5 = 3	2	2	[_, _, 30, 40, _]	—
`enqueue(50)`	(2 + 2) % 5 = 4	2	3	[_, _, 30, 40, 50]	—
`enqueue(60)`	(2 + 3) % 5 = 0	2	4	[60, _, 30, 40, 50]	—
`enqueue(70)`	(2 + 4) % 5 = 1	2	5	[60, 70, 30, 40, 50]	—
`dequeue()`	2	3	4	[60, 70, _, 40, 50]	30
`dequeue()`	3	4	3	[60, 70, _, _, 50]	40
`dequeue()`	4	0	2	[60, 70, _, _, _]	50
`dequeue()`	0	1	1	[_, 70, _, _, _]	60
`dequeue()`	1	2	0	[_, _, _, _, _]	70

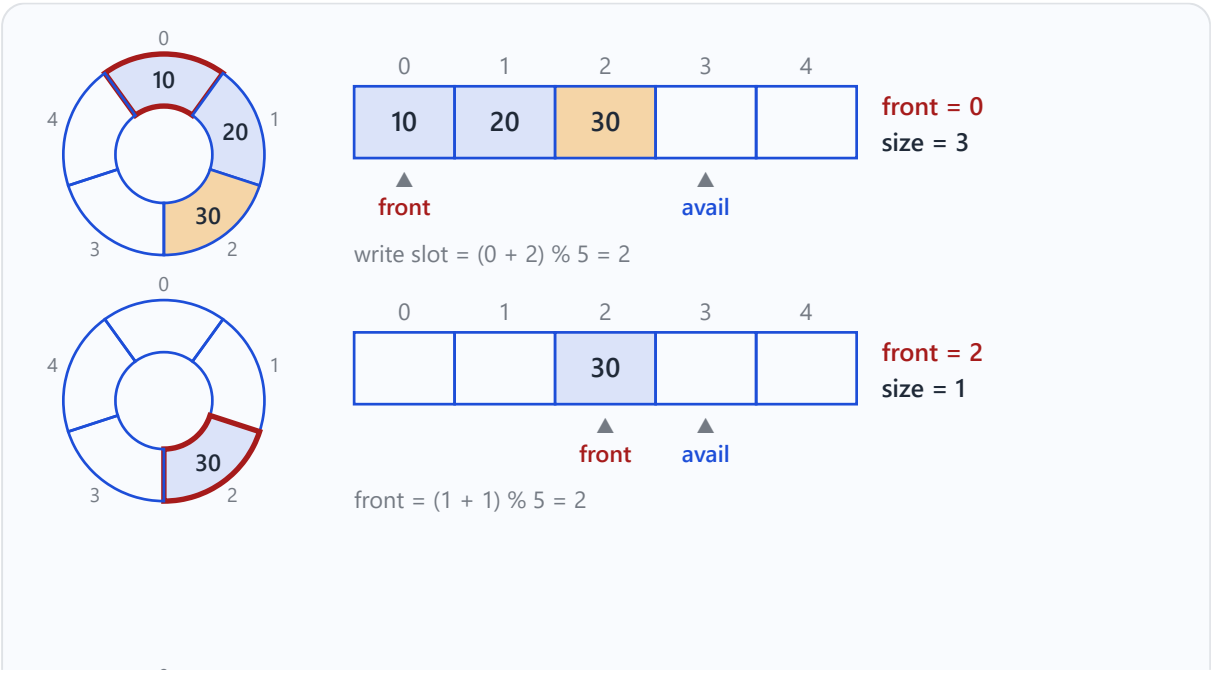
Table 8.6: Trace of a circular queue with *N = 5*. The array is written from index 0 to 4; "_" marks a free slot.

The first five rows match the simple array queue: three insertions fill slots 0–2, and two removals return 10 and 20 and move `front` to 2. Only `front` moves.

The difference appears at `enqueue(60)`: the insertion point *(2 + 3) % 5 = 0* has wrapped. Where the simple array queue would say "full," the circular queue reuses slot 0. `enqueue(70)` goes to slot 1 and `size` reaches 5; now the queue is truly full, with no slot wasted.

The next write position of the full queue, *(2 + 5) % 5 = 2*, is `front` itself; judged by indices alone the queue might look empty, but `size == 5` settles it.

Five removals then return 30, 40, 50, 60, 70. On the third, `front` becomes *(4 + 1) % 5 = 0*, so the front wraps too. After the last, `size = 0`. The output order equals the input order: the circular layout keeps FIFO intact.



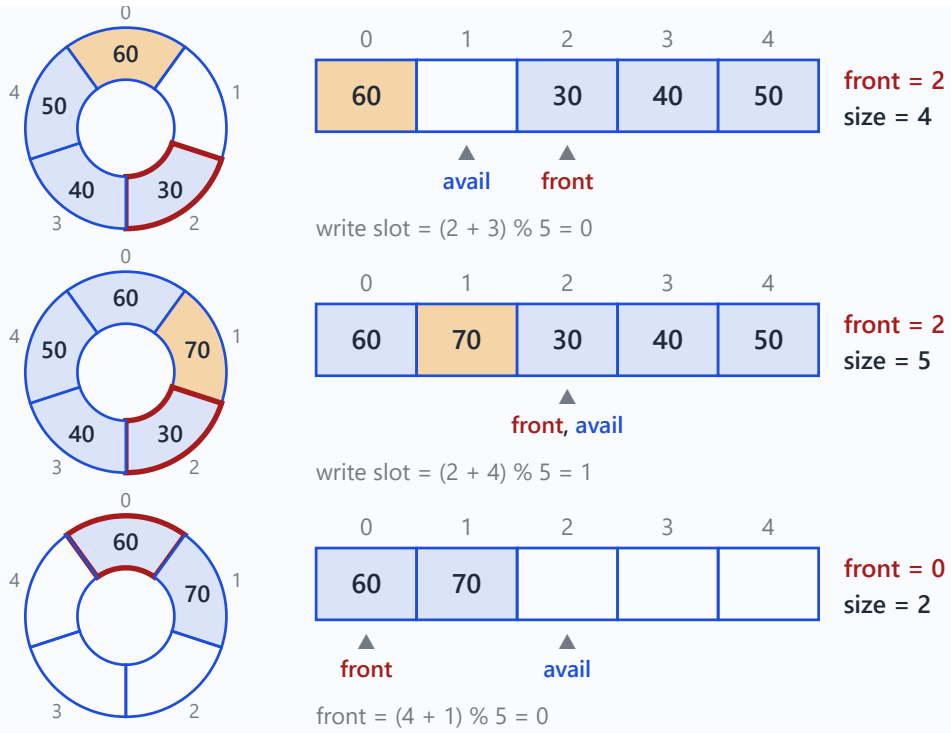


Figure 8.9: Five moments of Table 8.6, ring on the left and array on the right. The red-bordered slice is `front`; ▲ `avail` marks the next write position. (a) After three insertions. (b) After two removals, `front = 2`. (c) `enqueue(60)` has wrapped to slot 0. (d) After `enqueue(70)` the queue is full and `avail` meets `front`. (e) After three removals `front` has wrapped too: *(4 + 1) % 5 = 0*.

8.2.7 Code: the CircularQueue class

The class below holds the fields, constructor, and status queries of a counter-based circular queue; insertion and removal follow in Section 8.2.8.

```
public class CircularQueue<E> {
    private E[] data;        // stores the items
    private int front = 0;  // index of the first item
    private int size = 0;   // number of items

    @SuppressWarnings("unchecked")
    public CircularQueue(int capacity) {
        data = (E[]) new Object[capacity];
    }
    public int size()        { return size; }
    public boolean isEmpty() { return size == 0; }
    public boolean isFull()  { return size == data.length; }
    public E first()         { return isEmpty() ? null : data[front]; }
    // enqueue(), dequeue()
}
```

Line by line:

- `data` holds the items, `front` is the index of the first, and `size` is the count; both start at 0 because a new queue is empty.
- The constructor allocates the array. Java forbids `new E[capacity]`, because `E` is unknown at run time, so we create an `Object` array and cast it with `(E[])`. `@SuppressWarnings("unchecked")` silences the compiler's unchecked-cast warning for the constructor only; the cast is safe as long as the array never leaves the class.
- `size()` returns the count; `isEmpty()` tests `size == 0`, and `isFull()` tests `size == data.length`, the capacity.
- `first()` returns `null` for an empty queue and `data[front]` otherwise, changing nothing. `? :` is the conditional operator: the first value if the condition holds, the second if not.

The rear index is computed, `(front + size) % data.length`, rather than stored. Storing both `rear` and `size` would oblige every method to update both correctly, and a method that forgot one would leave them contradicting each other. Keeping each fact in one place removes that risk.

No method loops, so each is *O(1)*. The array has as many slots as the capacity, so space is *O(N)* however many items are stored.

8.2.8 Code: enqueue() and dequeue()

Here are the class's two basic methods:

```
public void enqueue(E item) {      // add at the rear
    if (isFull())
        throw new IllegalStateException("full");
    int avail = (front + size) % data.length;
    data[avail] = item;
    size++;
}
```

```
public E dequeue() {               // remove from the front
    if (isEmpty())
        throw new IllegalStateException("empty");
    E item = data[front];
    data[front] = null;            // drop the reference
    front = (front + 1) % data.length;
    size--;
    return item;
}
```

`enqueue` first checks for a full queue and throws `IllegalStateException("full")`, the code's way of reporting overflow. Otherwise it computes the insertion point into `avail`, writes the item, and increments `size`. `front` is untouched: adding at the rear does not affect the front.

`dequeue` throws `IllegalStateException("empty")` on underflow. Otherwise it saves the front item in `item`, writes `null` into the slot, advances `front`, decrements `size`, and returns the saved item.

`data[front] = null;` is not needed for correctness: `dequeue` returns the right item without it, and the slot is overwritten on the next lap anyway. It is about memory. The **garbage collector** frees only objects no reference points to, so an object that has left the queue but still sits in the array stays alive. With large items (images, file buffers) this becomes a real memory leak.

Line order matters too. `item = data[front]` must come *before* the `null` assignment, or the return value is lost, and `front` must advance *after* it, or the wrong slot is cleared.

Example 8.3

The two formulas at their boundary cases:

1. **Insertion point:** With $N = 5$, `front = 3`, `size = 2`, the items are in slots 3 and 4, and `avail = (3 + 2) % 5 = 5 % 5 = 0`: the new item wraps to slot 0.
2. **Advancing the front:** With $N = 5$ and `front = 4`, the front item is in the last slot; after a removal `front = (4 + 1) % 5 = 0`, so the front wraps too.

Each is one addition and one mod, with no loop or shifting: both methods are *O(1)*.

8.2.9 Code trace: enqueue()

We trace `enqueue` through a wrap. Initially `data.length = 5`, `front = 2`, `size = 2`, with 30 in slot 2 and 40 in slot 3; we add 50, 60, 70, and 80 using this method:

```
public void enqueue(E item) {
  if (isFull())
    throw new IllegalStateException("full");
  int avail = (front + size) % data.length;
  data[avail] = item;
  size++;
}
```

Call	`isFull()`	`avail`	`size` (after)	Array (after)
(start)	—	—	2	[_, _, 30, 40, _]
`enqueue(50)`	2 == 5? no	(2 + 2) % 5 = 4	3	[_, _, 30, 40, 50]
`enqueue(60)`	3 == 5? no	(2 + 3) % 5 = 0	4	[60, _, 30, 40, 50]
`enqueue(70)`	4 == 5? no	(2 + 4) % 5 = 1	5	[60, 70, 30, 40, 50]
`enqueue(80)`	5 == 5? **yes**	—	5	unchanged; exception: "full"

Table 8.7: Trace of the `enqueue` calls (`data.length = 5`; `front = 2` throughout).

For `enqueue(50)`, `size` (2) is below the capacity (5). The insertion point is *(2 + 2) % 5 = 4* — here mod changes nothing — so `data[4] = 50` and `size` becomes 3. `front` stays at 2.

For `enqueue(60)` the point is *(2 + 3) % 5 = 5 % 5 = 0*: the index wraps, and 60, logically last in the queue, sits physically first in the array. `enqueue(70)` writes to *(2 + 4) % 5 = 1* and raises `size` to 5.

For `enqueue(80)` the check gives `5 == 5`: `IllegalStateException("full")` is thrown and nothing changes. The capacity ran out at exactly 5 items, none wasted.

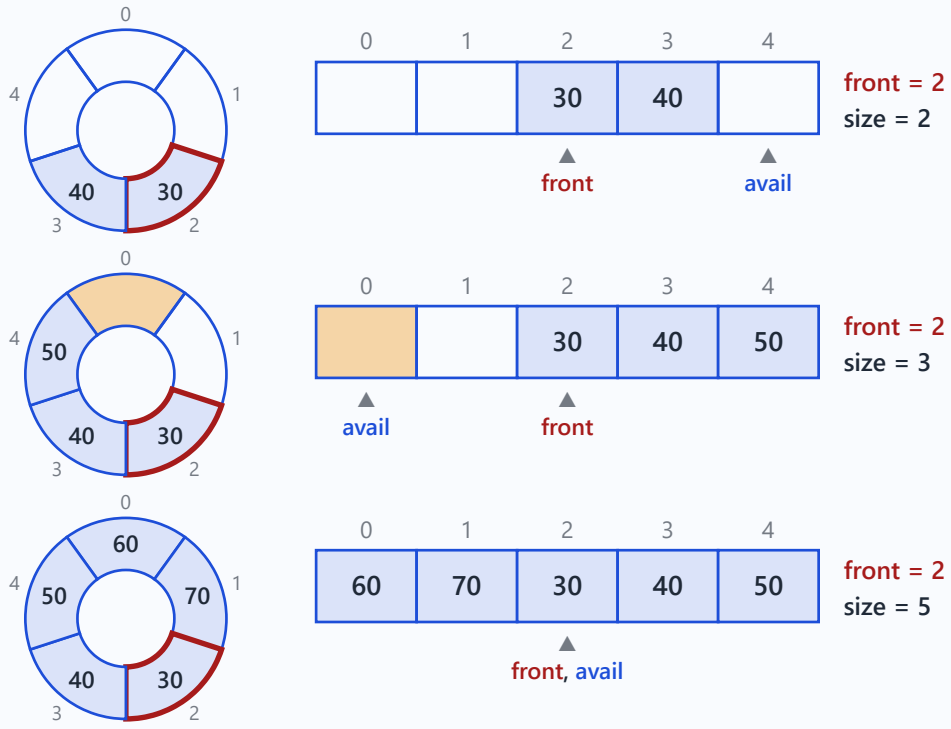


Figure 8.10: Three moments of the `enqueue` trace. (a) Start: 30 and 40, `front = 2`, `avail = 4`. (b) `enqueue(60)`: `avail = (2 + 3) % 5 = 0`, a wrap. (c) After `enqueue(70)`: full, `size = 5`, and `avail` lands on `front`.

8.2.10 Code trace: dequeue()

Starting from the full queue left by the previous trace (`front = 2`; 30, 40, 50, 60, 70 from front to rear), we call `dequeue()` three times.

```
public E dequeue() {
  if (isEmpty())
    throw new IllegalStateException("empty");
  E item = data[front];
  data[front] = null;          // drop the reference
  front = (front + 1) % data.length;
  size--;
  return item;
}
```

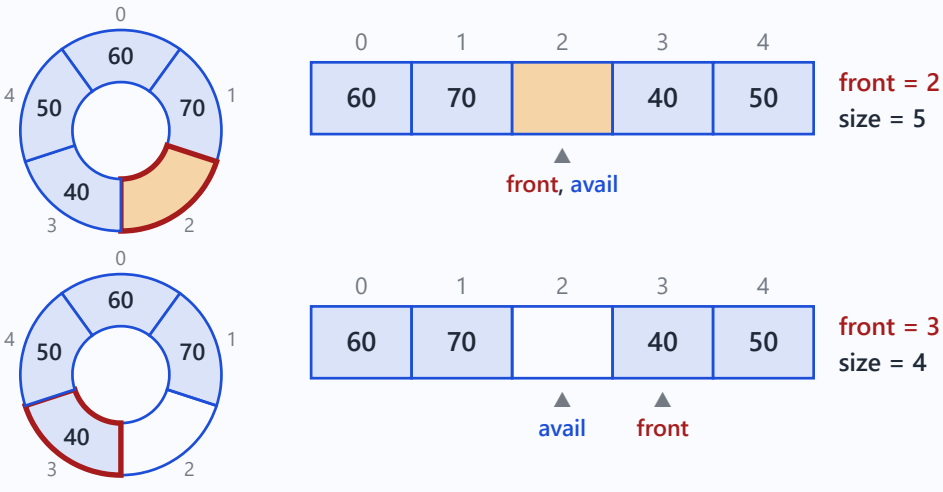
Call	`item`	Slot cleared	`front` (after)	`size` (after)	Array (after)
(start)	—	—	2	5	[60, 70, 30, 40, 50]
1st `dequeue()`	30	2	(2 + 1) % 5 = 3	4	[60, 70, _, 40, 50]
2nd `dequeue()`	40	3	(3 + 1) % 5 = 4	3	[60, 70, _, _, 50]
3rd `dequeue()`	50	4	(4 + 1) % 5 = 0	2	[60, 70, _, _, _]

Table 8.8: Trace of the `dequeue` calls (`data.length = 5`).

Each call takes the same five steps. `isEmpty()` is false, so `item = data[front]` takes the oldest item (`data[2] = 30` on the first call); `data[2] = null` clears the slot; `front` advances to *(2 + 1) % 5 = 3*; `size` drops to 4; and 30 is returned.

On the third call the front item is in the last slot, `data[4] = 50` , and `front` wraps: *(4 + 1) % 5 = 5 % 5 = 0*. Left are 60 (slot 0) and 70 (slot 1), now at the front of the queue both physically and logically.

No item moves and no loop runs: each call is *O(1)*.



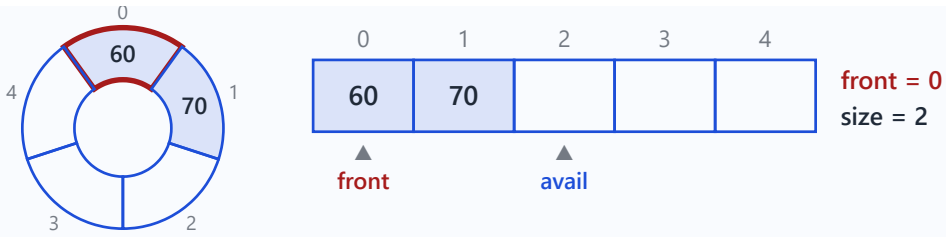


Figure 8.11: Three moments of the `dequeue` trace. (a) First call: `item = 30` taken and `data[2] = null` done; `front` and `size` not yet updated. (b) End of the first call: `front = 3`, `size = 4`, 30 returned. (c) After three calls: `front = 0`, `size = 2`; 60 and 70 remain.

8.2.11 Growing a full circular queue

A fixed-capacity queue fails when full. If the capacity is unknown in advance, an insertion into a full queue can instead allocate an array twice as large and move the items over:

```
private void resize(int newCap) {
  E[] temp = (E[]) new Object[newCap];
  for (int k = 0; k < size; k++)
    temp[k] = data[(front + k) % data.length];
  data = temp;
  front = 0;
}
public void enqueue(E item) {
  if (size == data.length) resize(2 * data.length);
  data[(front + size) % data.length] = item;
  size++;
}
```

`resize` copies the items to the start of the new array *in logical order*: item k, at old slot `(front + k) % data.length`, goes to new slot k. Then the old array is dropped and `front` becomes 0, since the front is now at the start. On a full queue, `enqueue` doubles the capacity first and then writes as usual.

Take a full queue of capacity 4 with `front = 2`: logically A, B, C, D, but physically C, D, A, B, since it has wrapped. Now `enqueue(E)` is called.

Step	Computation	New array (after)
Full?	`size == data.length` → 4 == 4, full	—
New array	`new Object[8]`	[␣, ␣, ␣, ␣, ␣, ␣, ␣, ␣]
`k = 0`	`temp[0] = data[(2 + 0) % 4] = data[2]` = A	[A, ␣, ␣, ␣, ␣, ␣, ␣, ␣]
`k = 1`	`temp[1] = data[(2 + 1) % 4] = data[3]` = B	[A, B, ␣, ␣, ␣, ␣, ␣, ␣]
`k = 2`	`temp[2] = data[(2 + 2) % 4] = data[0]` = C	[A, B, C, ␣, ␣, ␣, ␣, ␣]
`k = 3`	`temp[3] = data[(2 + 3) % 4] = data[1]` = D	[A, B, C, D, ␣, ␣, ␣, ␣]
`data = temp; front = 0`	the front is now in slot 0	[A, B, C, D, ␣, ␣, ␣, ␣]

Step	Computation	New array (after)
Write	`(0 + 4) % 8 = 4` → `data[4] = E` ; `size = 5`	[A, B, C, D, E, _, _, _]

Table 8.9: Trace of `resize(8)` during `enqueue(E)`. Old array `[C, D, A, B]`, `front = 2`.

At `k = 2` the mod kicks in, *(2 + 2) % 4 = 0*: the copy wraps around the old array and places C and D right behind A and B. The queue now lies in order from slot 0 with `front = 0`, and E goes to slot *(0 + 4) % 8 = 4*.

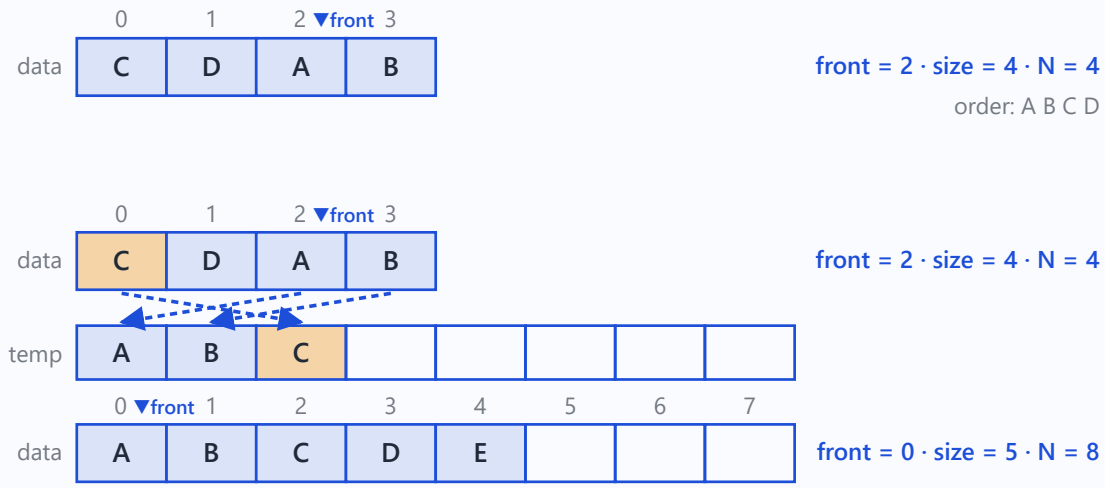


Figure 8.12: Growing a full circular queue. (a) Full: C, D, A, B in the array, `front = 2`, logical order A, B, C, D. (b) Step `k = 2`: C, from `data[0]`, goes to slot 2 of the new array. (c) After the resize and the write: A, B, C, D, E; `front = 0`, `size = 5`, *N = 8*.

Common mistake

Growing with `Arrays.copyOf(data, 2 * data.length)`. It copies each item to the *same index*, which works only if the queue has not wrapped. Above, it would give `[C, D, A, B, _, _, _, _]` with `front = 2`: after A and B the queue would take the empty slots 4 and 5 for items, and the wrapped C and D would drop out. Copy in `(front + k) % N` order and reset `front` to 0.

A resize copies every item and so costs *O(n)*, but doubling makes resizes rare, so it is more telling to look at the total cost of a long sequence of operations. This is **amortized analysis**.

Theorem 8.1 (Amortized cost of doubling)

If a circular queue doubles its capacity whenever it is full, n insertions into an initially empty queue make fewer than $3n$ writes and copies in total, so `enqueue` costs $O(1)$ amortized.

Proof. Let the initial capacity be 1; a larger one only means fewer copies. Resizes occur when the queue holds 1, 2, 4, ..., 2^k items, where 2^k is the largest power of two below n, so the copies total $1 + 2 + 4 + \ldots + 2^k = 2^{k+1} - 1$. Since $2^k < n$, this is under $2n - 1$, hence under $2n$. Adding the n writes, the cost stays below $2n + n = 3n$: fewer than 3 per insertion, a constant. Interleaved removals shrink the queue and so cannot add copies. □

For example, add 17 items to an empty queue of capacity 4 with no removals. Resizes at the 5th, 9th, and 17th insertions copy $4 + 8 + 16 = 28$ items; with 17 writes the total is 45, below $3 \cdot 17 = 51$. The theorem does not promise that every insertion is cheap: the 17th alone made 16 copies, and a single insertion can take $O(n)$. Real-time systems, where every operation must be quick, need a worst-case bound, not an amortized one.

Common mistake

Growing by a fixed amount (say, 10 slots) instead of doubling. Resizes then come at 10, 20, 30, ... items, the copies over n insertions add up to about $n^2/20$, and the total is $O(n^2)$, $O(n)$ per insertion. Amortized $O(1)$ requires growing by a constant factor.

8.3 Linked queues

An array-based queue has a preset capacity and can grow only by copying. A linked list creates a node per arriving item and never copies. This section builds a queue on a singly linked list and compares the two implementations.

8.3.1 The linked representation

Each item is a **node** holding the item's value (`data`) and a reference to the next node (`next`). The queue keeps `front` (first node), `rear` (last node), and the count `size` .

```
class Node {
    int data;
    Node next;
    Node(int data) {
        this.data = data;
    }
}
// fields:
Node front, rear;
int size;
```

Insertion happens at the end of the list and removal at its head. Each insertion allocates a node and each removal releases one, so there is no capacity limit and no "full queue" while memory lasts. The price is an extra reference per node and a new object per insertion.

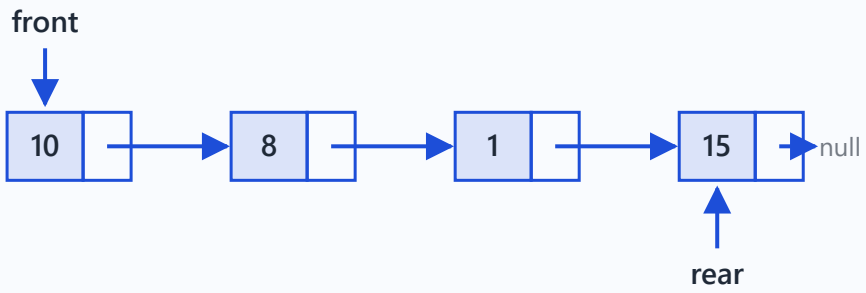


Figure 8.13: A linked queue: 10, 8, 1, 15 from front to rear. `front` points to the first node (10), `rear` to the last (15), whose `next` is `null`.

Why not the other way round? In a singly linked list, deleting the head is a single *O(1)* assignment, `front = front.next`, and adding at the end is easy thanks to `rear`: link the new node to `rear.next` and move `rear` to it. Deleting the last node, however, means finding the node before it, the new last node, and with no backward links that takes a walk from the head, *O(n)*. With the ends swapped, `dequeue` would be *O(n)*.

> **Common mistake**
>
> Keeping no `rear` reference. Every `enqueue` then walks the list to find the last node, costing *O(n)*, and filling a queue with *n* items costs $O(n^2)$. Two *O(1)* ends need two references.

8.3.2 The enqueue and dequeue algorithms

In any implementation the two operations follow the same steps:

`enqueue(x)`

1. Is the queue full? (Asked only for a fixed-capacity array.)
2. If so, report overflow and stop.
3. Otherwise, write `x` at the rear position.
4. Advance the rear to the next position.
5. Increment the item count and return successfully.

`dequeue()`

1. Is the queue empty?
2. If so, report underflow and stop.
3. Otherwise, take the front item.
4. Advance the front to the next item.
5. Decrement the item count and return the item taken.

Table 8.10 shows the corresponding code in the two implementations.

Step	Circular array	Linked list
Write at the rear	`data[(front + size) % N] = x`	`rear.next = temp; rear = temp`
Advance the front	`front = (front + 1) % N`	`front = front.next`
Full check	`size == N`	not needed (as long as memory lasts)

Table 8.10: The algorithm's steps in the two implementations.

In the circular array, advancing the rear is not a separate step: incrementing `size` moves the next write position. In the linked list, writing and advancing are two assignments: link the old last node to the new one, then move `rear`.

8.3.3 Code trace: enqueue

The linked queue's `enqueue`:

```
public void enqueue(int data) {
    Node temp = new Node(data);
    if (isEmpty()) {
        front = temp;
    } else {
        rear.next = temp;
    }
    rear = temp;
    size++;
}
```

`new Node(data)` returns an unlinked node holding the value, with `next` equal to `null`. If the queue is empty, the new node is also the first, so `front` points to it; otherwise the old last node's `next` is set to it. Either way it is now the last node, which is why `rear = temp` stands outside the `if` / `else`. Finally `size` grows by one.

Apply `enqueue(10)`, `enqueue(15)`, and `enqueue(20)` to an empty queue:

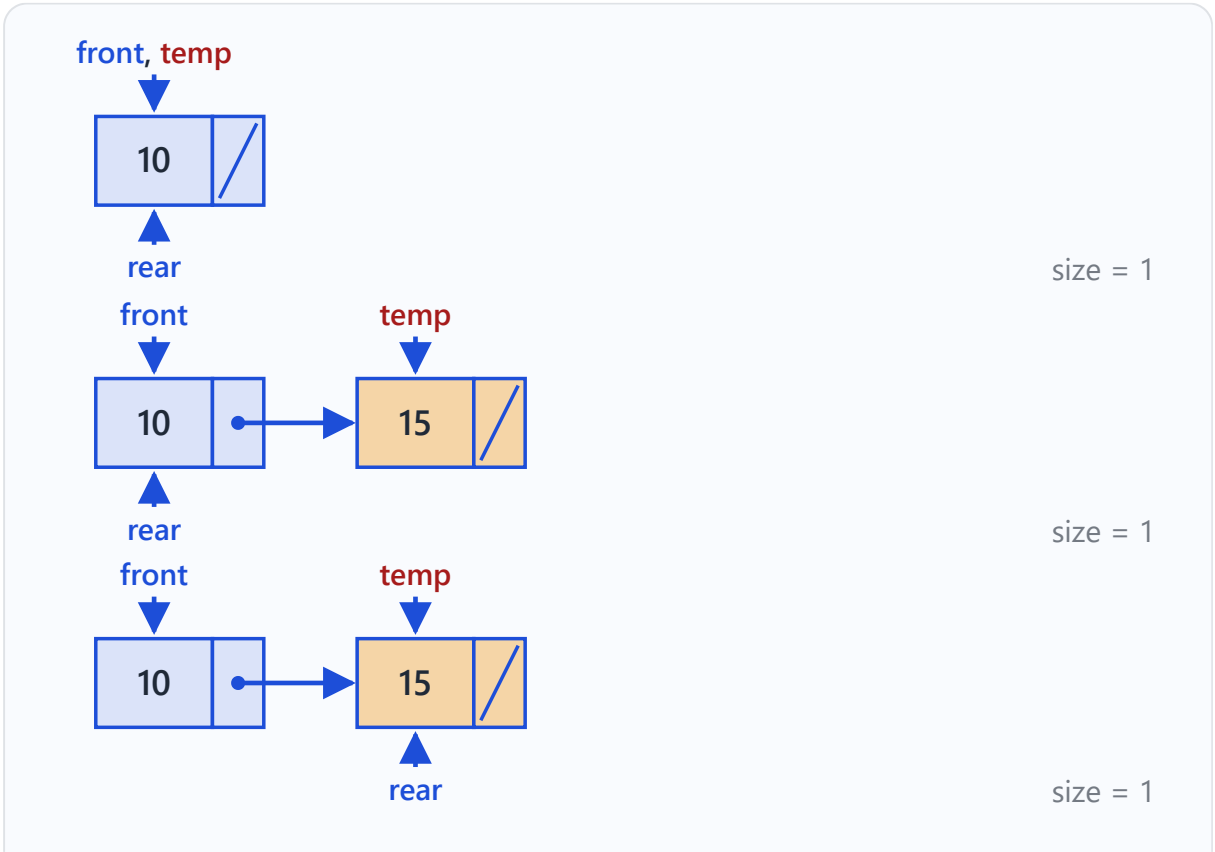
Call	`isEmpty()`	Branch taken	List (after)	`front`	`rear`	`size`
(start)	—	—	empty	`null`	`null`	0
`enqueue(10)`	`true`	`front = temp`	10	10	10	1
`enqueue(15)`	`false`	`rear.next = temp`	10 → 15	10	15	2
`enqueue(20)`	`false`	`rear.next = temp`	10 → 15 → 20	10	20	3

Table 8.11: Trace of the `enqueue` calls. The list is written from front to rear.

On the first call the queue is empty: `front = temp` and `rear = temp` make node 10 both front and rear.

On the second call the `else` branch links the old last node (10) to the new one (15) with `rear.next = temp`. This must happen *before* `rear` changes, since `rear` is the only way to reach the old last node. Then `rear = temp` moves the rear to 15; `front` stays. The third call repeats this for 20.

No call walks the list: each creates one node, assigns at most two references, and increments a counter, *O(1)*.



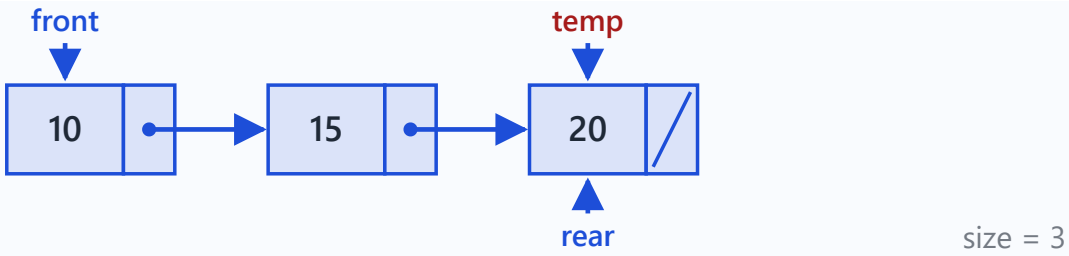


Figure 8.14: Four moments of the `enqueue` trace. (a) After the first call, node 10 is both `front` and `rear`. (b) Second call, `rear.next = temp`: 10 links to 15, but `rear` is still on 10. (c) `rear = temp`: the rear moves to 15. (d) After three calls: 10 → 15 → 20.

Common mistake

Putting `rear = temp` only in the `else` branch, or leaving it out. After the first insertion `front` points to the node but `rear` stays `null`, and the second insertion's `rear.next` throws `NullPointerException`. The new node is last in both cases, so the assignment belongs after the `if` / `else`.

8.3.4 Code trace: dequeue

The linked queue's `dequeue`:

```
public int dequeue() {
    if (isEmpty())
        throw new NoSuchElementException();
    int result = front.data;
    front = front.next;
    if (front == null)
        rear = null;
    size--;
    return result;
}
```

An empty queue throws `NoSuchElementException`. Otherwise the front value goes into `result` and `front` moves to the next node; the old first node is now unreferenced and the garbage collector reclaims it. `if (front == null) rear = null;` handles the edge case: if the removed node was the only one, the queue is now empty and `rear` must be cleared too. Finally `size` drops and `result` is returned.

Call `dequeue()` three times on the queue 10 → 15 → 20 from the previous trace:

Call	`result`	`front` (after)	`front == null` ?	`rear` (after)	`size`	List (after)
(start)	—	10	—	20	3	10 → 15 → 20

Call	`result`	`front` (after)	`front == null` ?	`rear` (after)	`size`	List (after)
1st `dequeue()`	10	15	`false`	20	2	15 → 20
2nd `dequeue()`	15	20	`false`	20	1	20
3rd `dequeue()`	20	`null`	`true`	`null`	0	empty

Table 8.12: Trace of the `dequeue` calls.

The first two calls take the front value and move `front` one node on; since `front` is not `null`, `rear` is untouched. Afterward both point to node 20.

The third call is the edge case. `result = 20`, and `front = front.next` makes `front` `null` while `rear` still points to the deleted node 20. `front == null` holds, so `rear = null` runs. Without it, an empty queue would keep `rear` on a deleted node: `rear()` would return a value not in the queue, and the node could not be freed.

The calls return 10, 15, 20, the insertion order, each in *O(1)*.

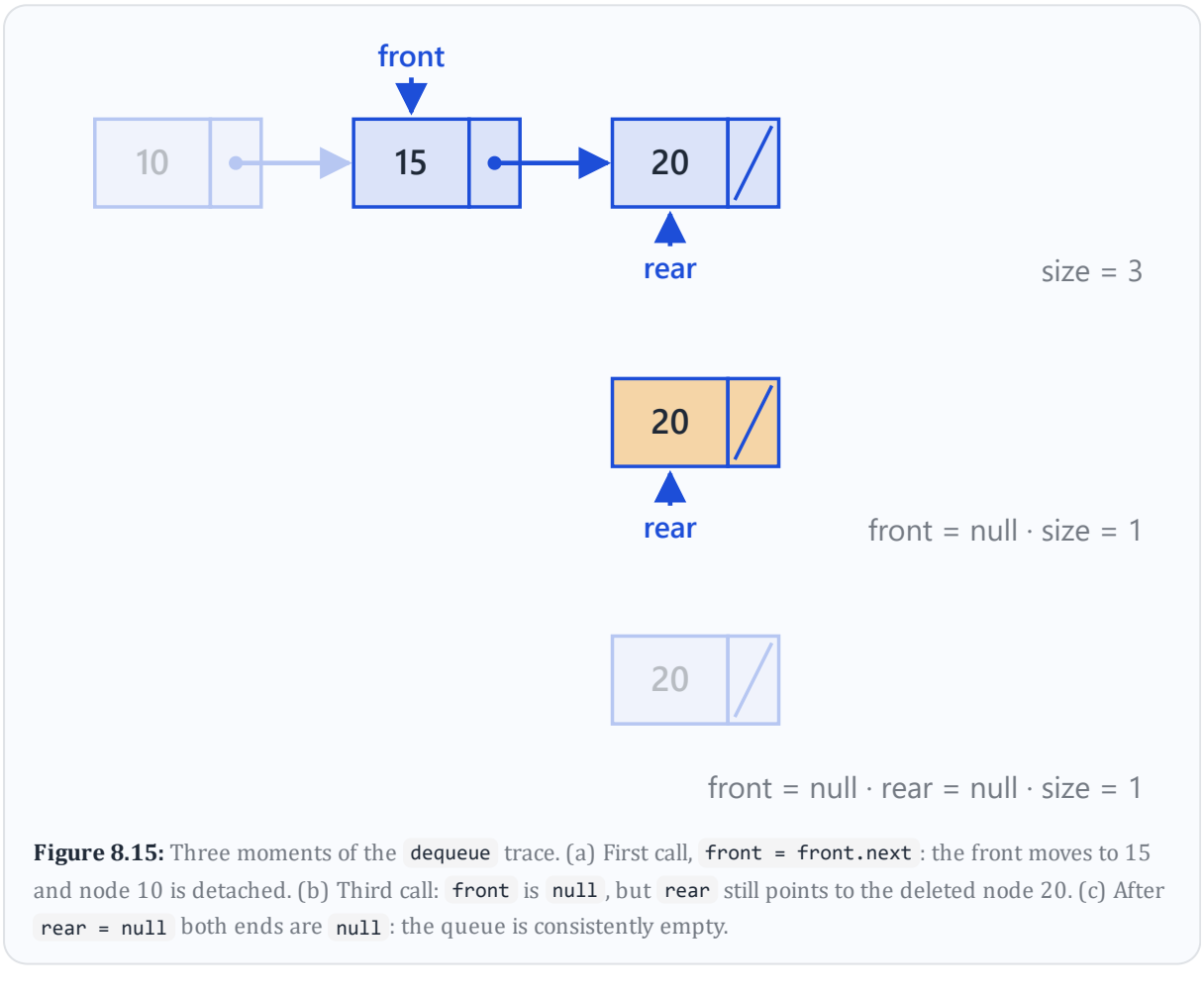


Figure 8.15: Three moments of the `dequeue` trace. (a) First call, `front = front.next`: the front moves to 15 and node 10 is detached. (b) Third call: `front` is `null`, but `rear` still points to the deleted node 20. (c) After `rear = null` both ends are `null`: the queue is consistently empty.

Common mistake

Not resetting `rear` when the last item leaves. The next `enqueue` sees an empty queue and resets both ends, so the bug often goes unnoticed — until `rear()` on an empty queue returns a wrong value. Bugs that appear only sometimes are the hardest to find.

8.3.5 Circular array or linked list?

Both implementations have *O(1)* basic operations; memory use and practical speed decide between them.

A **circular array** stores items contiguously. The processor reads memory in blocks, so contiguous data is faster to access than scattered nodes; it is *cache-friendly*, and it needs no per-item reference. But the capacity is fixed: when it runs out, the queue reports overflow or copies into a bigger array, a copy that is *O(n)* alone and *O(1)* amortized. The array also occupies its full capacity even when nearly empty.

A **linked list** has no capacity limit, uses memory in proportion to its items, and makes every operation *O(1) in the worst case*, since nothing is ever copied in bulk. But each node carries an extra reference and each insertion creates an object, and allocation plus later garbage collection cost more than writing an array slot.

Operation	Circular array	Linked list
`enqueue`	O(1)	O(1)
`dequeue`	O(1)	O(1)
`first` / `size` / `isEmpty`	O(1)	O(1)
Memory	O(N) (the capacity)	O(n) (the item count)

Table 8.13: Costs of the two queue implementations (*n*: number of items, *N*: capacity).

In practice the circular array usually wins, which is why Java's `ArrayDeque` uses it. A linked list pays off when every operation must be guaranteed fast or when the queue's length swings widely.

8.4 Kinds of queues and the deque

The basic queue fixes one end for insertion and one for removal. Other kinds of queues relax or change that rule; this section surveys them and studies the most general, the double-ended queue.

8.4.1 Kinds of queues

Kind	Insert	Remove	Note
Simple queue	rear	front	FIFO
Input-restricted	rear only	front and rear	restricted deque
Output-restricted	front and rear	front only	restricted deque
Circular	rear	front	array slots are reused
Double-ended (deque)	front and rear	front and rear	not bound to FIFO
Priority	anywhere	highest priority	ordered by priority

Table 8.14: Kinds of queues.

The circular queue defines no new behavior: it is an implementation of the simple queue, and its users see no difference. The other kinds change the ADT.

The **double-ended queue**, which inserts and removes at both ends, is the most general; **input-restricted** and **output-restricted** queues limit it. An input-restricted queue inserts only at the rear but removes at either end, so it can act as a queue (removing at the front) or a stack (removing at the rear). An output-restricted queue accepts items at both ends but removes only at the front: an urgent job can jump to the front, an ordinary one joins at the back.

A **priority queue** ignores arrival order: removal always yields the item of highest priority, as an emergency room sees patients by urgency rather than arrival. It is usually built on a heap and studied separately; despite its name, it is not FIFO.

> **Common mistake**
>
> Confusing **deque** with `dequeue`. Deque, short for *double-ended queue* and pronounced "deck," is a data structure; `dequeue` is an operation.

> **Common mistake**
>
> Taking Java's `PriorityQueue` for FIFO. In `Queue<Integer> q = new PriorityQueue<>();` the variable's type is `Queue`, but the object is a priority queue: after `q.offer(5); q.offer(1); q.offer(3);`, `poll()` returns 1, then 3, because under natural ordering the smallest item has the highest priority.

8.4.2 The double-ended queue (deque)

Definition 8.3

A **double-ended queue** (**deque**) is an abstract data type that keeps items in a sequence and allows insertion and removal at both the front and the rear.



Figure 8.16: A deque holding 27, 26, 22, 21, 20, 23, 24, 25 from front to rear; items can be added and removed at the front (head) and at the rear (tail).

A deque is not bound to FIFO; its behavior depends on the ends its client uses. Adding and removing at the rear only makes it a **stack** (LIFO); adding at the rear and removing at the front makes it a **queue** (FIFO). It subsumes both, which is why Java recommends `ArrayDeque` for either role.

Figure 8.16 shows the flexibility. If 20 came first, then 21 and 22 may have gone to the front, 23 and 24 to the rear, 26 and 27 to the front, and 25 to the rear. The result is neither arrival order nor its reverse; each item's place depends on the end it entered.

Table 8.15 lists our operation names and their `ArrayDeque` counterparts.

Operation	Java `ArrayDeque`	Time
`addFirst(e)`	`offerFirst(e)`	O(1)
`addLast(e)`	`offerLast(e)`	O(1)
`removeFirst()`	`pollFirst()`	O(1)
`removeLast()`	`pollLast()`	O(1)
`first()` / `last()`	`peekFirst()` / `peekLast()`	O(1)

Table 8.15: Deque operations.

`ArrayDeque` also has stack operations: `push(e)` adds at the front (`addFirst`) and `pop()` removes there (`removeFirst`); its queue operations `offer` and `poll` are `offerLast` and `pollFirst`. Mixing them on one object confuses: `push` puts an item at the *front*, `offer` at the *rear*.

A deque can use a circular array — inserting at the front steps back with `front = (front - 1 + N) % N` and writes there, and removing at the rear just decrements `size` — as `ArrayDeque` does. These notes use the other option, a doubly linked list.

Example 8.4

A deque can test whether a word is a palindrome, reading the same backward: add the letters at the rear, then remove one from each end at a time and compare.

```
boolean isPalindrome(String s) {
    Deque<Character> d = new ArrayDeque<>();
    for (char c : s.toCharArray())
        d.offerLast(c);
    while (d.size() > 1)
        if (!d.pollFirst().equals(d.pollLast()))
            return false;
    return true;
}
```

1. `"kayak"`: The deque holds k, a, y, a, k. Round one removes k and k, round two a and a; both match. Only y remains, `size() > 1` fails, and the result is `true`.
2. `"karma"`: Round one removes k and a, which differ, so the method returns `false` at once.

Each letter is added once and removed at most once: *O(n)* time and *O(n)* extra space. The comparison uses `equals`, because `pollFirst()` and `pollLast()` return `Character` objects and `!=` would compare identities, not values.

8.4.3 Deque operations step by step

Our deque is a **doubly linked list**: each node has `prev` and `next` references, and the deque keeps `head` (first node), `tail` (last node), and `size`.

Apply `addLast(5)`, `addLast(8)`, `addFirst(3)`, `addLast(9)`, `removeLast()`, and `removeFirst()` to an empty deque:

Operation	References changed	List (after)	`size`	Returns
(start)	—	empty; `head = tail = null`	0	—
`addLast(5)`	`head = tail = node`	5	1	—
`addLast(8)`	`node.prev = tail; tail.next = node; tail = node`	5 ⇄ 8	2	—
`addFirst(3)`	`node.next = head; head.prev = node; head = node`	3 ⇄ 5 ⇄ 8	3	—
`addLast(9)`	(add at the rear)	3 ⇄ 5 ⇄ 8 ⇄ 9	4	—
`removeLast()`	`tail = tail.prev; tail.next = null`	3 ⇄ 5 ⇄ 8	3	9

Operation	References changed	List (after)	`size`	Returns
`removeFirst()`	`head = head.next; head.prev = null`	5 ⇄ 8	2	3

Table 8.16: Trace of the deque operations.

The first insertion is special: in an empty list the new node is both head and tail. Later a new node is linked to the end node *in both directions*; adding at the rear sets the new node's `prev` to the old tail and the old tail's `next` to the new node, and adding at the front mirrors this.

With both ends referenced, every insertion is *O(1)*. The removals show what `prev` is worth: `removeLast()` finds the new tail in one step, `tail.prev`, where a singly linked list would need an *O(n)* walk from the head to the next-to-last node. The price is one more reference per node.

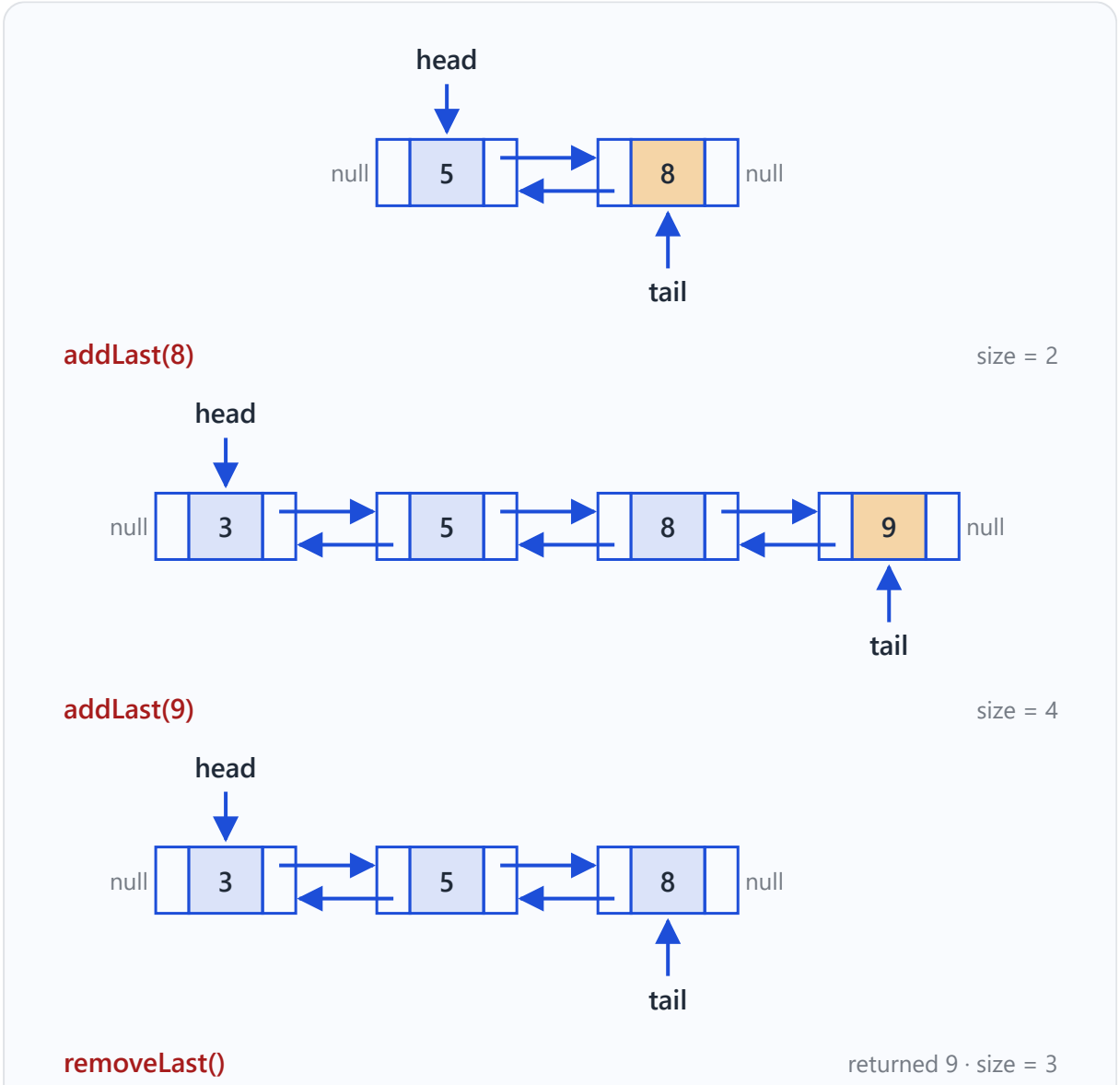


Figure 8.17: Three moments of Table 8.16. (a) After two `addLast` calls: 5 ⇄ 8, each link two-way. (b) After `addFirst(3)` and `addLast(9)` : 3 ⇄ 5 ⇄ 8 ⇄ 9, `size = 4` . (c) `removeLast()` has returned 9; `tail` is back

on 8.

8.4.4 Code: adding at either end

The node class:

```
class Node<E> {
    E data;
    Node<E> prev, next;
    Node(E data) { this.data = data; }
}
```

The insertion methods:

```
public void addFirst(E data) {
    Node<E> node = new Node<>(data);
    if (isEmpty()) {
        head = tail = node;
    } else {
        node.next = head;
        head.prev = node;
        head = node;
    }
    size++;
}
```

```
public void addLast(E data) {
    Node<E> node = new Node<>(data);
    if (isEmpty()) {
        head = tail = node;
    } else {
        node.prev = tail;
        tail.next = node;
        tail = node;
    }
    size++;
}
```

Both methods create a node with `prev` and `next` set to `null`. In an empty deque it becomes head and tail at once: `head = tail = node;` runs right to left, assigning `tail` first and then `head`. Otherwise three assignments follow.

In `addFirst` the new node points to the old head (`node.next = head`), the old head points back (`head.prev = node`), and the head moves (`head = node`). The order matters: doing `head = node` first would lose the old head before it could be linked.

Swapping `head` with `tail` and `prev` with `next` turns `addFirst` into `addLast`. Neither loops; both are *O(1)*.

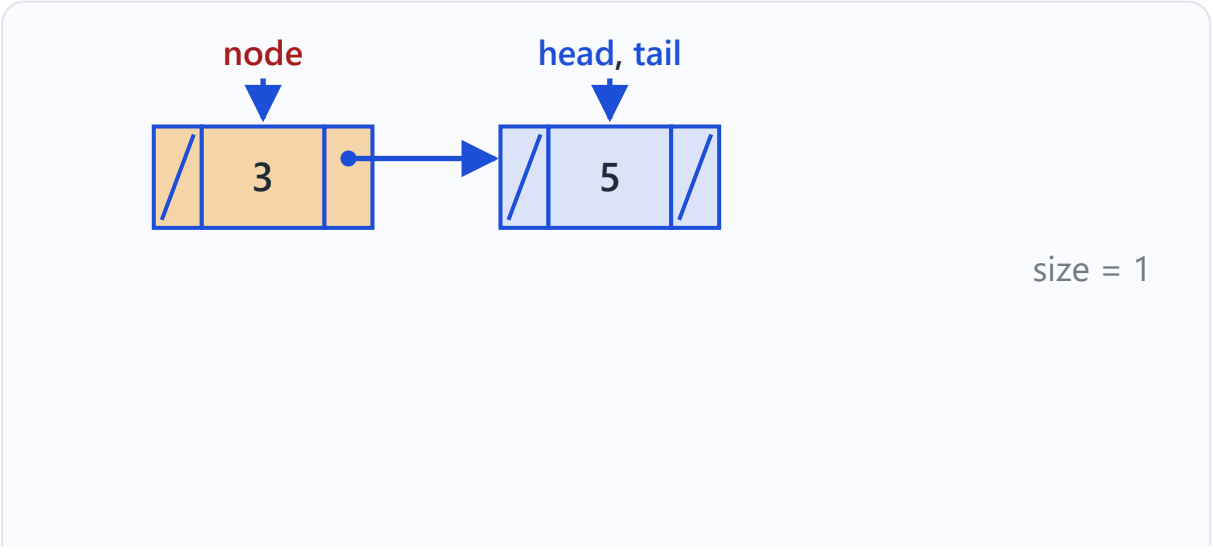
8.4.5 Code trace: addFirst()

Apply `addFirst(5)` and then `addFirst(3)` to an empty deque; 3 should end up in front.

Line	Effect	`head`	`tail`	`size`
`addFirst(5)`				
`new Node<>(5)`	`prev = next = null`	`null`	`null`	0
`isEmpty()`	`true`	`null`	`null`	0
`head = tail = node`	a single node forms both ends	5	5	0
`size++`		5	5	1
`addFirst(3)`				
`new Node<>(3)`	`prev = next = null`	5	5	1
`isEmpty()`	`false`	5	5	1
`node.next = head`	the new node points to the old head (5)	5	5	1
`head.prev = node`	back link: 5 points to the new node	5	5	1
`head = node`	the head reference moves to the new node	3	5	1
`size++`		3	5	2

Table 8.17: Trace of the `addFirst` calls.

The list is now 3 ⇄ 5, read 3, 5 from head to tail and 5, 3 from tail to head. Two-way traversal relies on both `node.next = head` and `head.prev = node`. Nothing is traversed: *O(1)*.



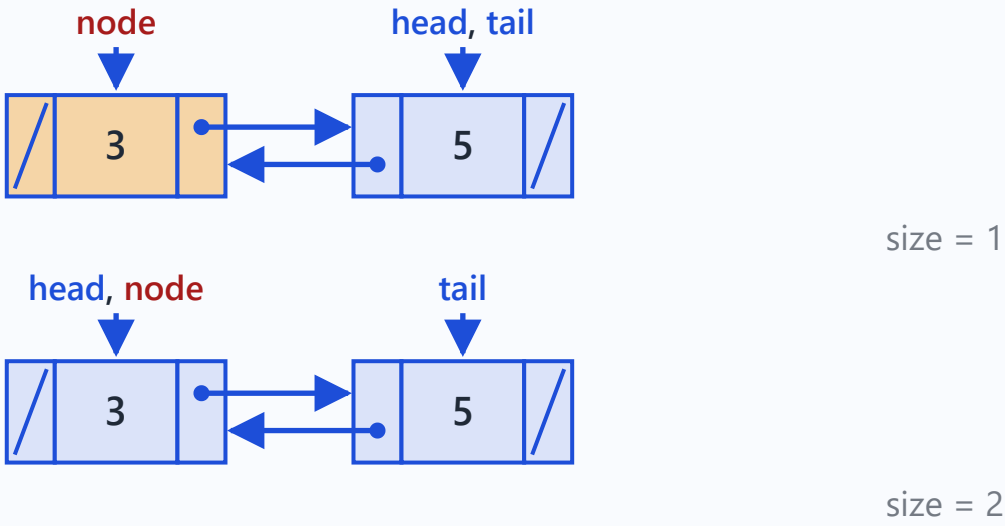


Figure 8.18: Three moments of `addFirst(3)`. (a) `node.next = head`: the new node points to 5, which does not point back yet. (b) `head.prev = node`: the link is two-way; `head` is still on 5. (c) After `head = node` and `size++`: 3 ⇄ 5, `size = 2`.

8.4.6 Code: removing from either end

```
public E removeFirst() {
    if (isEmpty()) return null;
    E data = head.data;
    head = head.next;
    if (head == null)
        tail = null;       // now empty
    else
        head.prev = null;
    size--;
    return data;
}
```

```
public E removeLast() {
    if (isEmpty()) return null;
    E data = tail.data;
    tail = tail.prev;
    if (tail == null)
        head = null;       // now empty
    else
        tail.next = null;
    size--;
    return data;
}
```

On an empty deque both methods return `null`, as in Goodrich's interface. Otherwise the end value goes into `data` and the end reference moves one node inward: `head = head.next` or `tail = tail.prev`.

If that reference is now `null`, the removed node was the only one, so the other end is set to `null` too. Otherwise the new end's outward link is cut (`head.prev = null` or `tail.next = null`), so the list no longer refers to the removed node. Then `size` drops and the value is returned.

> **Common mistake**
>
> Not clearing both ends when the last item goes. If `removeLast` omits `head = null`, `head` still points to the deleted node while `tail` is `null`: the node cannot be freed, and `first()` returns a value not in the deque. It is the linked queue's `rear = null` bug again.

8.4.7 Code trace: removeLast()

Call `removeLast()` twice on the deque 3 ⇄ 5: 5 comes out, then 3, leaving it empty.

Line	Effect	`head`	`tail`	`size`
1st `removeLast()`				
`isEmpty()`	`false` (size 2)	3	5	2
`data = tail.data`	`data = 5`	3	5	2
`tail = tail.prev`	the tail steps back to 3	3	3	2
`tail == null` ?	`false`: the deque is not empty	3	3	2
`tail.next = null`	the link to 5 is cut	3	3	2
`size--`; `return`	returns 5	3	3	1
2nd `removeLast()`				
`isEmpty()`	`false` (size 1)	3	3	1
`data = tail.data`	`data = 3`	3	3	1
`tail = tail.prev`	there was a single node: `tail = null`	3	`null`	1
`tail == null` ?	`true`: the deque is now empty	3	`null`	1
`head = null`	both ends `null`	`null`	`null`	1
`size--`; `return`	returns 3	`null`	`null`	0

Table 8.18: Trace of the `removeLast` calls.

On the first call, `tail = tail.prev` finds the next-to-last node without searching. `tail` is not `null`, so the `else` branch cuts the link to node 5 with `tail.next = null`, detaching it.

On the second call the single node's `prev` is `null`, so `tail` becomes `null` while `head` still points to node 3. `tail == null` holds and `head = null` runs; without it `head` would keep pointing to the deleted node. Both calls are *O(1)*.

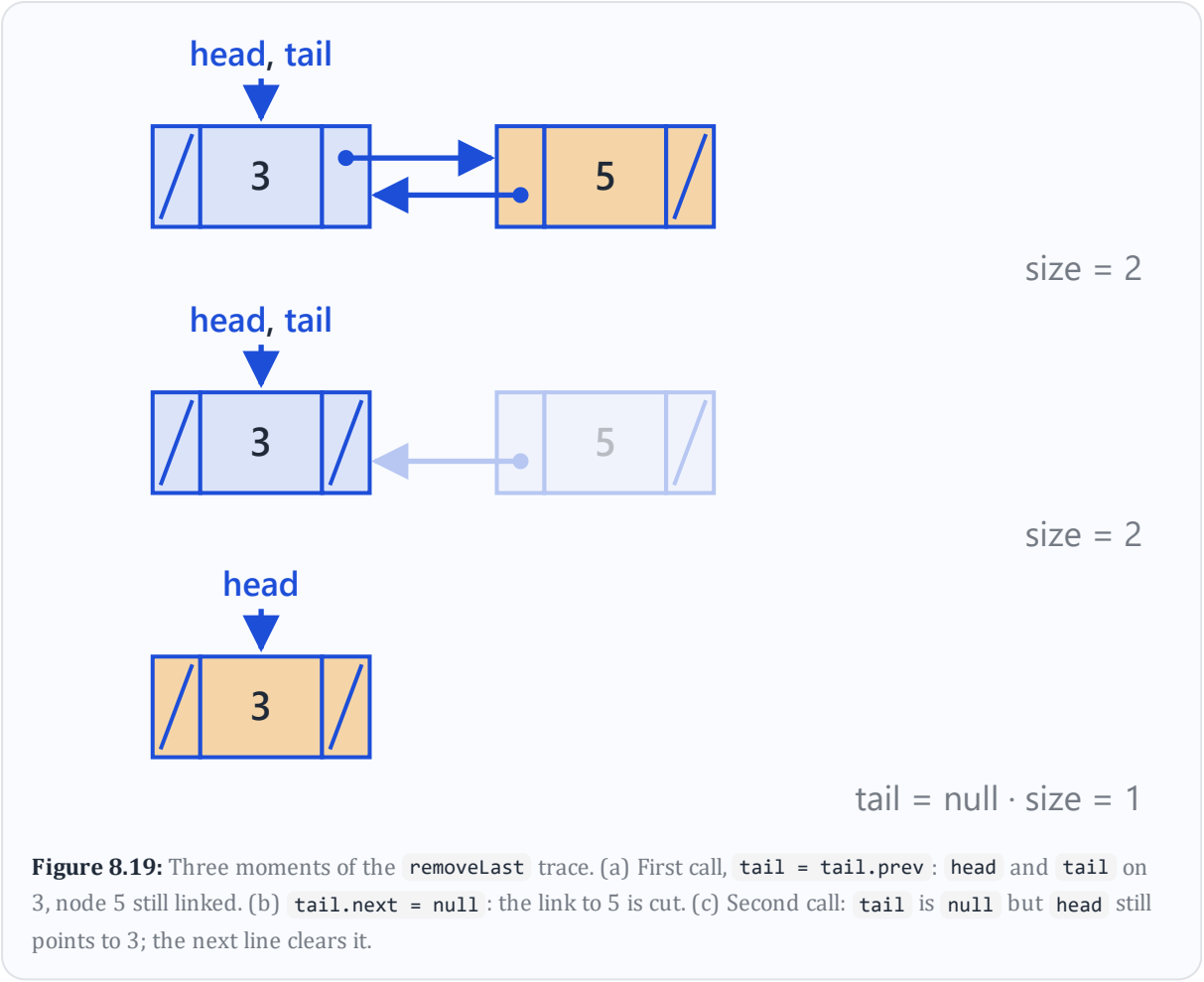


Figure 8.19: Three moments of the `removeLast` trace. (a) First call, `tail = tail.prev`: `head` and `tail` on 3, node 5 still linked. (b) `tail.next = null`: the link to 5 is cut. (c) Second call: `tail` is `null` but `head` still points to 3; the next line clears it.

8.4.8 Sliding window maximum

A classic problem that needs both ends of a deque: slide a window of width k across an array `a` from left to right and report the largest value at each position. This is a **sliding window** problem. For `a = {1, 3, -1, -3, 5, 3, 6}` and $k = 3$ the five windows {1, 3, −1}, {3, −1, −3}, {−1, −3, 5}, {−3, 5, 3}, {5, 3, 6} have maxima 3, 3, 5, 5, 6.

Brute force scans k items in each of the $n - k + 1$ windows, $O(n \cdot k)$, which reaches $O(n^2)$ for $k = n/2$. Doing better means reusing earlier work. The key fact: an item with a greater or equal item to its right can never again be a window maximum, since it leaves the window first while the larger item stays. Such items can be discarded at once.

The algorithm keeps candidate *indices* in a deque whose values decrease from front to rear, so the maximum is always in front. For each new index i:

1. **Drop from the front** the front index if it has left the window ($\leq i - k$).
2. **Drop from the rear** every index whose value is at most `a[i]`; it is no longer a candidate.
3. **Add** i at the rear. From the first full window on ($i \geq k - 1$), the value at the front index is the window's maximum.

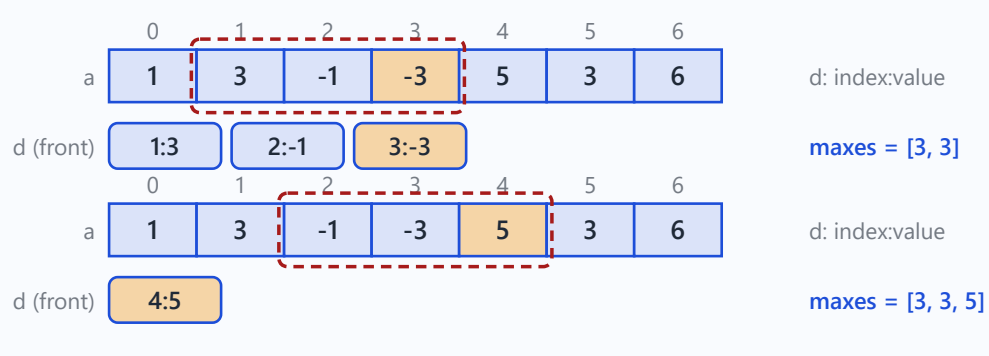
```
int[] windowMax(int[] a, int k) {
  int[] maxes = new int[a.length - k + 1];
  Deque<Integer> d = new ArrayDeque<>(); // indices
  for (int i = 0; i < a.length; i++) {
    if (!d.isEmpty() && d.peekFirst() <= i - k)
      d.pollFirst();        // outside the window
    while (!d.isEmpty() && a[d.peekLast()] <= a[i])
      d.pollLast();         // no longer a candidate
    d.offerLast(i);
    if (i >= k - 1) maxes[i - k + 1] = a[d.peekFirst()];
  }
  return maxes;
}
```

`d` holds indices, and values are read as `a[d.peekFirst()]` and `a[d.peekLast()]`. Whether an item has left the window depends on its position; with values alone we could not tell which of several equal values had left. `maxes` holds the $n - k + 1$ results, the window ending at i writing to `maxes[i - k + 1]`.

i	`a[i]`	Dropped	`d` (after)	`maxes`
0	1	—	0:1	—
1	3	from the rear: 0:1	1:3	—
2	−1	—	1:3, 2:−1	[3]
3	−3	—	1:3, 2:−1, 3:−3	[3, 3]
4	5	from the front: 1:3 (outside the window); from the rear: 3:−3, 2:−1	4:5	[3, 3, 5]
5	3	—	4:5, 5:3	[3, 3, 5, 5]
6	6	from the rear: 5:3, 4:5	6:6	[3, 3, 5, 5, 6]

Table 8.19: Trace of `windowMax({1, 3, -1, -3, 5, 3, 6}, 3)`. `d` is written front to rear as "index:value."

The key step is $i = 4$. The window covers indices 2–4, so the front index 1 ($1 \le 4 - 3 = 1$) is dropped. The incoming 5 beats −3 and −1 at the rear, which can never be maxima again, so they go too, leaving 4:5. At $i = 3$, by contrast, −3 was added and nothing dropped: small as it is, −3 could still become the maximum once 3 and −1 left.



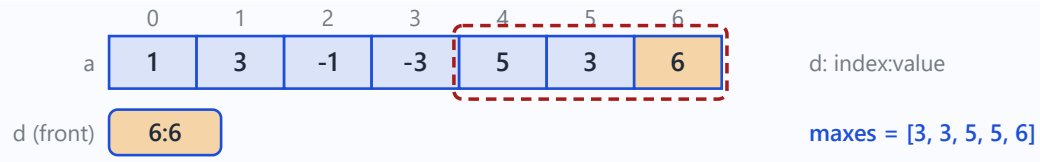


Figure 8.20: Three moments of the sliding window trace; the dashed frame is the window. (a) *i* = 3: `d` = 1:3, 2:−1, 3:−3, values decreasing. (b) *i* = 4: 1 dropped from the front, 3 and 2 from the rear; only 4:5 remains. (c) *i* = 6: 6 has dropped both rear candidates; result [3, 3, 5, 5, 6].

Theorem 8.2 (Cost of the sliding window)

On an array of *n* items, `windowMax` runs in *O(n)* time with *O(k)* extra space.

Proof. The inner `while` loop may run often for one *i*, so counting per loop misleads. Count per index instead: each enters the deque exactly once (`offerLast`) and leaves at most once, from the front (`pollFirst`) or the rear (`pollLast`). That is *n* insertions and at most *n* removals, each *O(1)*, so *O(n)* in all. Only indices inside the window can be in the deque, so the extra space is at most *k*. □

This counting trick — each item enters once and leaves at most once — recurs throughout queue and stack algorithms; the amortized analysis of the two-stack queue uses it too.

Common mistake

Forgetting to flip the comparison when adapting the code for the sliding window *minimum*. Then the values must *increase* from front to rear, and the rear-drop test becomes `a[d.peekLast()] >= a[i]`.

8.5 Applications of queues

In the first three problems below, a queue cycles through work fairly or walks a structure level by level; the last builds a queue from stacks.

8.5.1 A round-robin scheduler

Many processes want to run, but a processor core runs one at a time. The **scheduler** shares the processor among them, and one of the simplest and fairest methods is **round robin**.

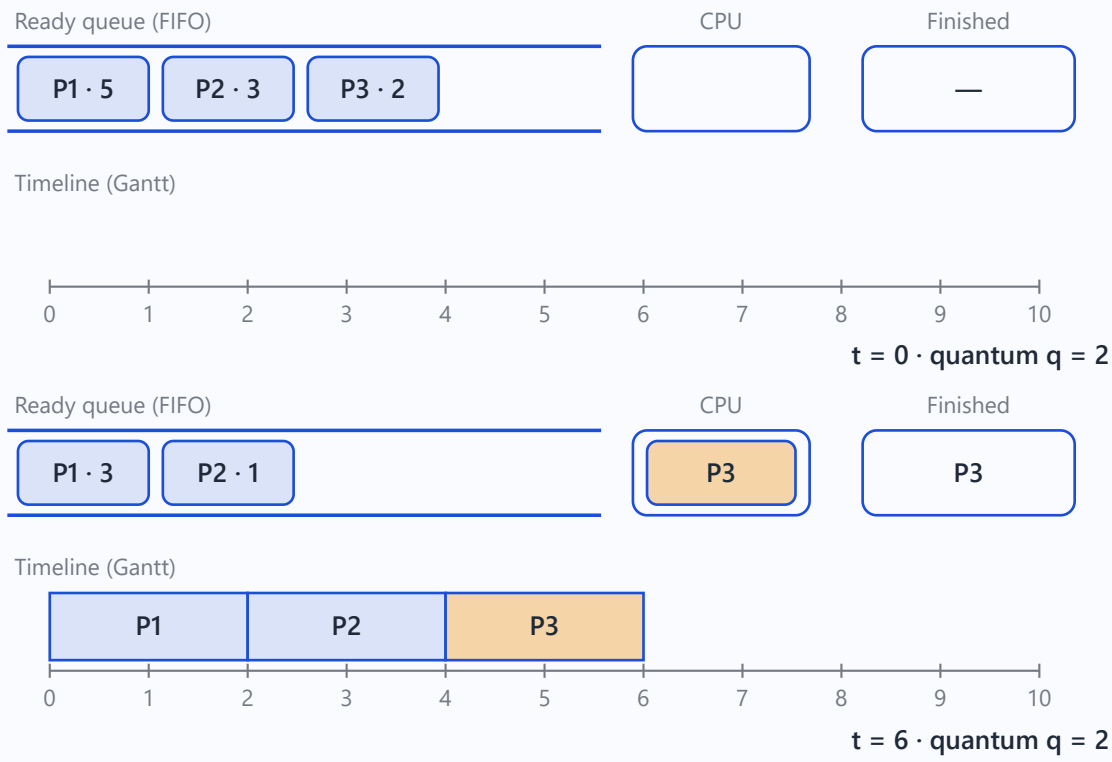
Ready processes wait in a FIFO *ready queue*. The front process runs for at most *q* time units, the **time slice** or **quantum**. If it finishes, it leaves; if not, it returns to the rear with its remaining time. No process can monopolize the processor, and each runs again within one round.

Example 8.5

The ready queue holds P1, P2, P3 with 5, 3, and 2 units left, and $q = 2$. When does each finish?

1. **t = 0–2:** P1 runs 2 units; 3 left, back to the rear. Queue: P2, P3, P1.
2. **t = 2–4:** P2 runs 2 units; 1 left. Queue: P3, P1, P2.
3. **t = 4–6:** P3's 2 units fit in one slice; it finishes at $t = 6$. Queue: P1, P2.
4. **t = 6–8:** P1 runs 2 more units; 1 left. Queue: P2, P1.
5. **t = 8–9:** P2's last unit; it finishes at $t = 9$, and the processor moves on without waiting out the slice. Queue: P1.
6. **t = 9–10:** P1 finishes at $t = 10$; the queue is empty.

Completion times: P3 = 6, P2 = 9, P1 = 10. The shortest process finished first and the longest last, though P1 had entered first.



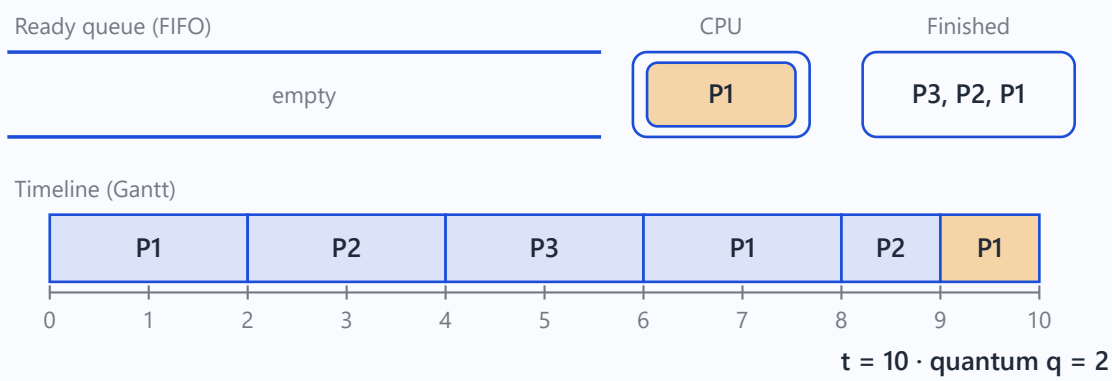


Figure 8.21: Three moments of Example 8.5: ready queue, CPU, and finished processes above, the timeline (Gantt chart) below. (a) *t = 0*: P1 · 5, P2 · 3, P3 · 2. (b) *t = 6*: P3 done; P1 · 3 and P2 · 1 remain. (c) *t = 10*: all done, in the order P3, P2, P1.

Performance hinges on *q*. Too large, and every process finishes in one slice: the method becomes first-come, first-served, with short processes stuck behind long ones. Too small, and the processor switches constantly; each context switch has a cost, so much of its time goes to switching instead of work. In practice *q* is long enough for most processes to finish an interactive task in one slice.

A process's waiting time is its completion time minus its running time: *10 − 5 = 5* for P1, *9 − 3 = 6* for P2, *6 − 2 = 4* for P3, an average of *15/3 = 5*. Run back to back (P1, P2, P3), they would finish at 5, 8, and 10 and wait 0, 5, and 8, an average of *13/3 ≈ 4.33*. So round robin does not always lower the average wait; what it buys is that no process waits long for its first turn — all three ran within the first 6 units.

8.5.2 Code trace: roundRobin()

The scheduler's code is short. A process is represented by this class:

```
class Process {
    String name;      // process name (P1, P2, …)
    int remaining;    // remaining running time
    Process(String name, int remaining) { this.name = name; this.remaining = remaining; }
}
```

```
void roundRobin(Queue<Process> q, int quantum) {
  int t = 0;
  while (!q.isEmpty()) {
    Process p = q.poll();
    int run = Math.min(quantum, p.remaining);
    t += run;
    p.remaining -= run;
    if (p.remaining > 0) q.offer(p);
    else System.out.println(p.name + " t=" + t);
  }
}
```

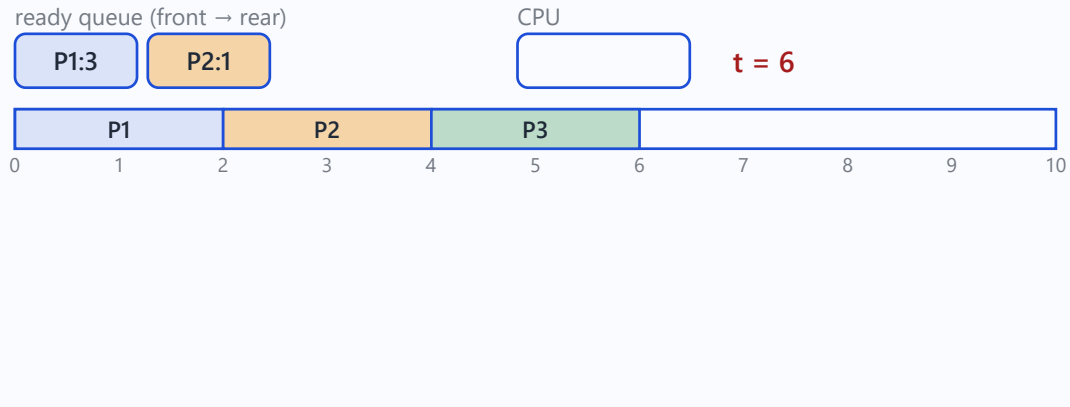
`t` is the elapsed time, and the loop runs until the queue is empty. Each round takes the front process with `poll()` and runs it for `run = Math.min(quantum, p.remaining)`, so a process about to finish runs only for what it needs. `t` grows and `p.remaining` shrinks by `run`; a process with time left returns to the rear with `offer(p)`, and a finished one has its completion time printed.

Round	`p`	`run`	`t`	`p.remaining`	Outcome	Queue (after)
(start)	—	—	0	—	—	P1:5, P2:3, P3:2
1	P1	min(2, 5) = 2	2	3	back to the rear	P2:3, P3:2, P1:3
2	P2	min(2, 3) = 2	4	1	back to the rear	P3:2, P1:3, P2:1
3	P3	min(2, 2) = 2	6	0	"P3 t=6"	P1:3, P2:1
4	P1	min(2, 3) = 2	8	1	back to the rear	P2:1, P1:1
5	P2	min(2, 1) = 1	9	0	"P2 t=9"	P1:1
6	P1	min(2, 1) = 1	10	0	"P1 t=10"	empty

Table 8.20: Trace of `roundRobin(q, 2)` (P1 · 5, P2 · 3, P3 · 2). The queue is written front to rear as "name:remaining."

The code repeats the hand computation. In rounds 5 and 6, `run` is 1 because the remaining time is shorter than the slice. The output is "P3 t=6", "P2 t=9", "P1 t=10".

A round makes one `poll` and at most one `offer`, so it is *O(1)*. A process of length *s* takes *⌈s/q⌉* rounds; here *⌈5/2⌉ + ⌈3/2⌉ + ⌈2/2⌉ = 3 + 2 + 1 = 6*.



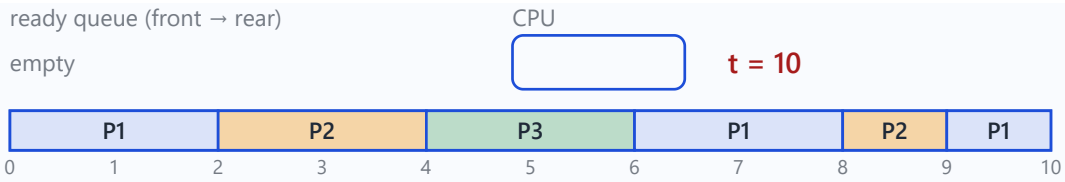


Figure 8.22: Two moments of the `roundRobin` trace. (a) End of round 3: P3 done ($t = 6$); P1 and P2 remain. (b) End of the loop: the queue is empty, and the timeline reads P1, P2, P3, P1, P2, P1.

Common mistake

Advancing `t` by a full slice (`t += quantum`) even when less time remains. Every round then counts 2 units: P3 still ends at 6, but P2 seems to end at 10 instead of 9 and P1 at 12 instead of 10, because idle processor time is counted as work. The increment must be `Math.min(quantum, p.remaining)` .

8.5.3 The Josephus problem

The Josephus problem comes from an old puzzle. *n* people stand in a circle and count off from a given person; every *k*th person leaves the circle, and counting resumes with the next one. Who is the last one left?

A queue rotates the circle naturally: moving the front person to the rear advances the count by one. Each round moves $k - 1$ people to the rear and then eliminates the person at the front, the *k*th.

```
String josephus(Queue<String> q, int k) {
  while (q.size() > 1) {
    for (int i = 1; i < k; i++)
      q.offer(q.poll());      // front to the rear
    String out = q.poll();
    System.out.println(out + " eliminated");
  }
  return q.poll();
}
```

The `for` loop runs `q.offer(q.poll())` for `i = 1` to `k - 1`, moving the front person to the rear each time. Then `poll()` removes the front person for good and prints the name. The outer `while` runs while more than one person remains, and the survivor is returned.

Here is the trace for $n = 5$ (A, B, C, D, E) and $k = 3$.

Round	Moved to the rear	Queue (before removal)	Eliminated	Queue (after)
(start)	—	—	—	A, B, C, D, E
1	A, B	C, D, E, A, B	C	D, E, A, B

Round	Moved to the rear	Queue (before removal)	Eliminated	Queue (after)
2	D, E	A, B, D, E	A	B, D, E
3	B, D	E, B, D	E	B, D
4	B, D	B, D	B	D

Table 8.21: Trace of `josephus(q, 3)` ($n = 5$). The queue is written front to rear.

The elimination order is C, A, E, B; D survives. In round 4 both remaining people move to the rear, the count comes back to B, and B goes. As the circle shrinks the count may lap it several times, and the queue handles this with no special case.

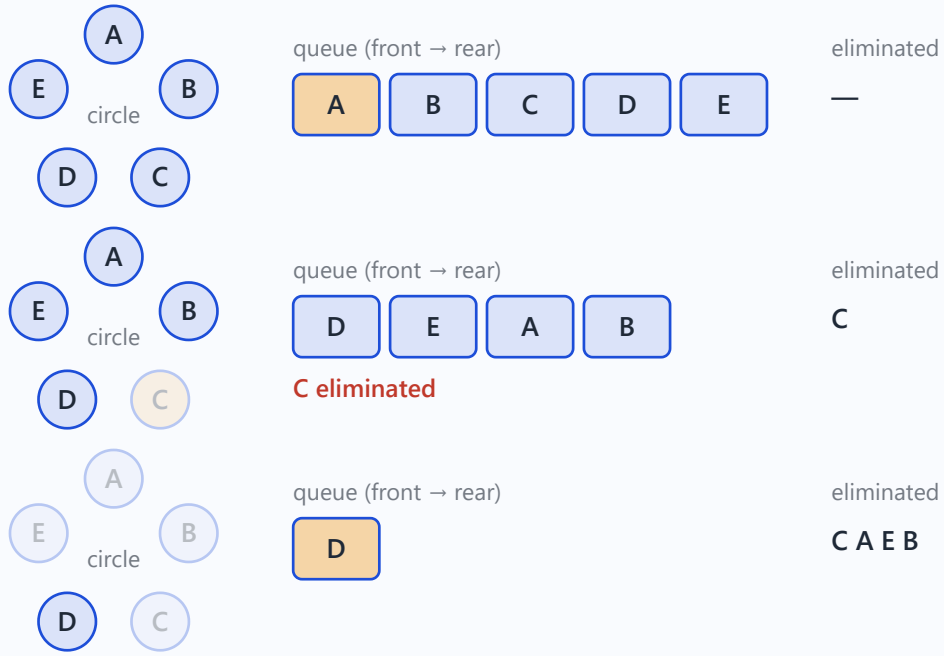


Figure 8.23: Three moments of the Josephus trace: circle, queue, eliminated. (a) Start: A, B, C, D, E. (b) Round 1: A and B moved to the rear, C eliminated; queue D, E, A, B. (c) End: C, A, E, B eliminated; D survives.

Of n people, $n - 1$ are eliminated, each round costing $k - 1$ moves and one removal, k queue operations. The total is $(n - 1) \cdot k$, or $O(n \cdot k)$; the example used $4 \cdot 3 = 12$.

If only the survivor matters, a recurrence suffices. Numbering from 0, $J(1) = 0$ and $J(n) = (J(n - 1) + k) \% n$; for $k = 3$, $J(2) = 1$, $J(3) = 1$, $J(4) = 0$, $J(5) = 3$, and person 3 is the fourth, D. This runs in $O(n)$ but gives no elimination order, which the queue plays out step by step.

8.5.4 Binary numbers from 1 to n

The task: list the binary representations of 1 to n in increasing order — {"1", "10", "11"} for $n = 3$, {"1", "10", "11", "100", "101"} for $n = 5$.

Converting each number separately works, but a queue is more elegant. Appending "0" to the binary form of x gives $2x$, appending "1" gives $2x + 1$: "10" (2) yields "100" (4) and "101" (5). This arranges the numbers in a tree rooted at "1," each node's children being the node with "0" and "1" appended.

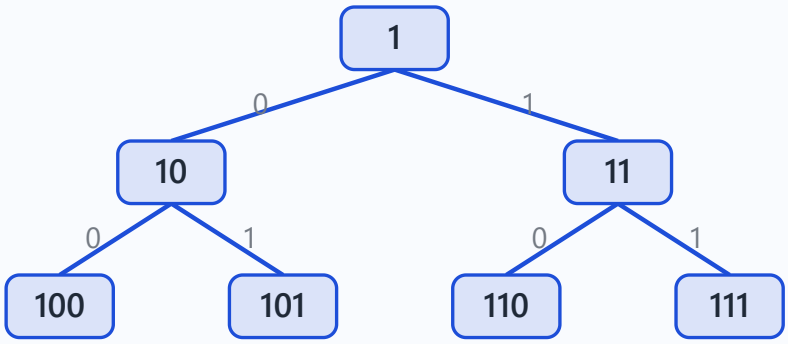


Figure 8.24: The binary number tree: 1 has children 10 and 11, 10 has 100 and 101, 11 has 110 and 111. Read level by level, left to right, it lists the numbers in increasing order: 1, 10, 11, 100, 101, 110, 111.

A queue walks a tree level by level: take a string, write it to the result, add its two children at the rear. First in is first out, so each level is finished before the next begins, left to right. This is **breadth-first traversal**. Numbers on one level have equal length and grow left to right, and deeper ones are longer and thus larger, so the output is exactly increasing.

```
String[] generateBinary(int n) {
    String[] result = new String[n];
    Queue<String> q = new LinkedList<>();
    q.offer("1");
    for (int i = 0; i < n; i++) {
        result[i] = q.poll();
        q.offer(result[i] + "0");
        q.offer(result[i] + "1");
    }
    return result;
}
```

8.5.5 Code trace: generateBinary(4)

The method creates a result array of n slots and an empty queue — a `LinkedList`, used only through `Queue` operations — and adds the root "1." Each of the n rounds removes the front string, writes it to the result, and adds its two children.

`i`	`poll()`	Added at the rear	Queue (after)	`result`
(start)	—	"1"	1	[␣, ␣, ␣, ␣]
0	"1"	"10", "11"	10, 11	[1, ␣, ␣, ␣]

`i`	`poll()`	Added at the rear	Queue (after)	`result`
1	"10"	"100", "101"	11, 100, 101	[1, 10, _, _]
2	"11"	"110", "111"	100, 101, 110, 111	[1, 10, 11, _]
3	"100"	"1000", "1001"	101, 110, 111, 1000, 1001	[1, 10, 11, 100]

Table 8.22: Trace of `generateBinary(4)`. The queue is written front to rear.

The front string is always the smallest waiting. At `i = 4`, *4 < 4* fails, and the method returns {"1", "10", "11", "100"}. The five strings left in the queue go unused, a small waste that a counter limiting the insertions could avoid.

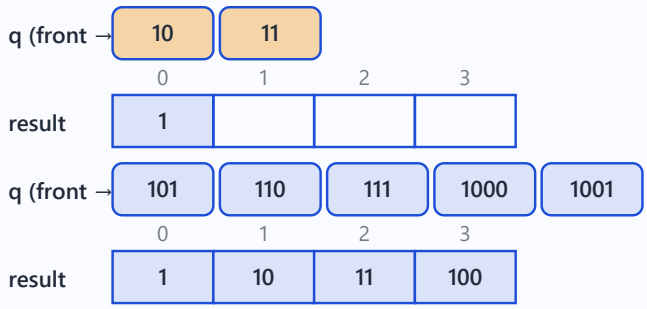


Figure 8.25: Two moments of the `generateBinary(4)` trace. (a) After round one: "1" is in the result, its children "10" and "11" in the queue. (b) End: `result` = {1, 10, 11, 100}; 101, 110, 111, 1000, 1001 remain queued.

In queue operations the cost is *n* `poll` and *2n* `offer` calls, *O(n)*. But each operation builds a string of about $\log_2 n$ characters, and concatenation is linear in length, so the character-level work is *O(n log n)*. The queue holds at most about *n + 1* strings.

8.5.6 A queue from two stacks

A design question to finish: can stacks alone make a queue? One stack reverses order, so it cannot; but reversing twice restores the order, and that is the idea behind two stacks.

Insertions go onto `inbox`; removals come from `outbox`. `enqueue` pushes onto `inbox`. `dequeue` first checks `outbox`: if it is empty, every item in `inbox` is popped and pushed onto `outbox`, which reverses the order and puts the oldest item on top; then `outbox` is popped.

```
Deque<Integer> inbox = new ArrayDeque<>();
Deque<Integer> outbox = new ArrayDeque<>();
void enqueue(int x) {
  inbox.push(x);
}
int dequeue() {
  if (outbox.isEmpty())
    while (!inbox.isEmpty())
      outbox.push(inbox.pop());
  return outbox.pop();
}
```

`ArrayDeque` serves as the stack, with `push` and `pop` at its head. On an empty queue both stacks are empty, the transfer moves nothing, and `outbox.pop()` throws `NoSuchElementException`: underflow is reported by exception.

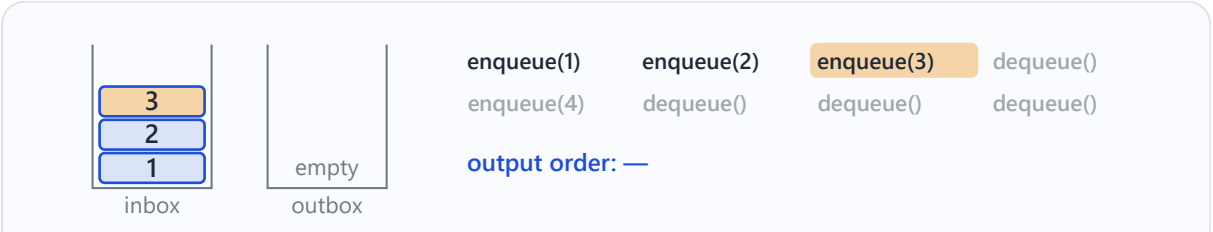
Apply `enqueue(1)`, `enqueue(2)`, `enqueue(3)`, `dequeue()`, `enqueue(4)`, and three `dequeue()` calls; stacks are written bottom to top.

Operation	Transfer	`inbox` (after)	`outbox` (after)	Returns
`enqueue(1)`	—	1	empty	—
`enqueue(2)`	—	1, 2	empty	—
`enqueue(3)`	—	1, 2, 3	empty	—
`dequeue()`	3, 2, 1 moved (3 items)	empty	3, 2	1
`enqueue(4)`	—	4	3, 2	—
`dequeue()`	none (`outbox` not empty)	4	3	2
`dequeue()`	none	4	empty	3
`dequeue()`	4 moved (1 item)	empty	empty	4

Table 8.23: Trace of the two-stack queue. The stacks are written bottom to top.

On the first `dequeue`, `outbox` is empty. The 3 on top of `inbox` moves first, landing at the bottom of `outbox`, then 2, then 1, which ends on top and is popped. FIFO holds: the first item in is the first out.

`enqueue(4)` puts 4 into `inbox` and leaves 2 and 3 in `outbox`. The next two `dequeue` calls find `outbox` non-empty, so 2 and 3 come out without a transfer; 4 rightly waits, having arrived after them. The last `dequeue` finds `outbox` empty and transfers and removes 4. The output 1, 2, 3, 4 matches the input; 4 items were transferred in all.



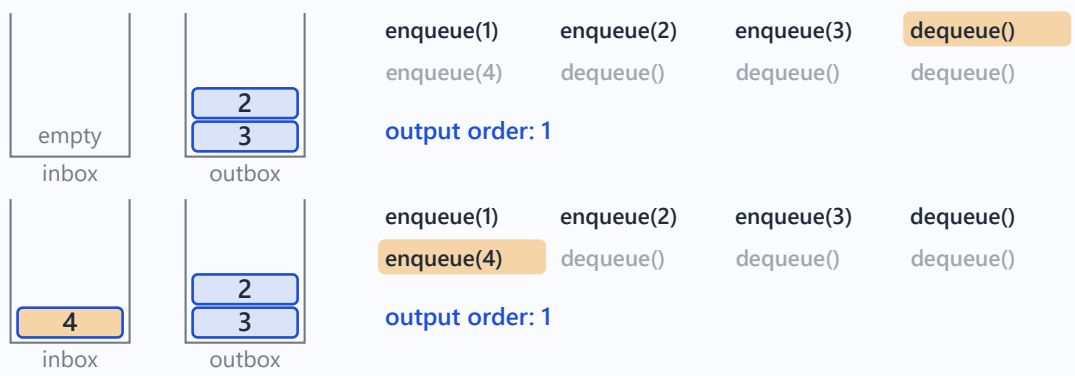


Figure 8.26: Three moments of the two-stack queue. (a) After three `enqueue` calls, `inbox` holds 1, 2, 3 bottom to top. (b) First `dequeue`: the transfer reversed the order, 1 was popped, and 3, 2 remain in `outbox`. (c) `enqueue(4)`: 4 goes into `inbox`; `outbox` is untouched.

One `dequeue` can be costly: with `outbox` empty and n items in `inbox`, the transfer makes n pops and n pushes, $O(n)$ in the worst case. But such calls cannot be frequent.

Theorem 8.3 (Amortized cost of the two-stack queue)

Starting from an empty queue, any sequence of m `enqueue` / `dequeue` operations costs $O(m)$, so both operations are $O(1)$ amortized.

Proof. Follow one item. It is pushed onto `inbox` once; in a transfer it is popped from `inbox` and pushed onto `outbox` once, and it never returns to `inbox`, so it is transferred at most once; finally it is popped from `outbox` once. That is at most 4 stack operations per item, and m operations add at most m items, so the total is at most $4m$. □

The 8 operations of Table 8.23 made 4 pushes onto `inbox`, 4 transfers (a pop and a push each), and 4 pops from `outbox`: $4 + 8 + 4 = 16$ stack operations, exactly the bound $4 \cdot 4 = 16$, since each item was transferred once.

Common mistake

Transferring on every `dequeue`, even when `outbox` is not empty. This is slow and wrong: in the example, 4 transferred right after `enqueue(4)` would sit on top of `outbox` and leave before 2. Transfer only when `outbox` is empty.

Conversely, one queue can make a stack. `push(x)` adds `x` and then moves every earlier item from the front to the rear, one at a time, bringing `x` to the front, so `pop` removes from the front. Here `pop` is $O(1)$ and `push` $O(n)$, and amortizing does not help, because every `push` rotates the whole queue.

8.6 Common mistakes

Most queue bugs fall into a few types. The table collects the mistakes noted throughout the chapter; use it as a checklist when reviewing queue code.

Mistake	What happens?	The fix
Forgetting `rear` on the last item	In a linked queue, removing the last node makes `front = null`, but `rear` keeps pointing to the deleted node.	`if (front == null) rear = null;`
Full / empty confusion	With only `front` and `rear`, `front == rear` holds for both the empty and the full queue.	A `size` counter or one empty slot.
Forgetting the mod	In a wrapped queue the index runs off the array: `ArrayIndexOutOfBoundsException`.	`(front + size) % N`; going backward, `(front - 1 + N) % N`.
`Arrays.copyOf` when growing	In a wrapped circular queue the logical order is scrambled and items are lost.	Copy items in `(front + k) % N` order and set `front = 0`.
Mixing up `add` / `offer`, `remove` / `poll`	`add`, `remove`, and `element` throw; `offer`, `poll`, and `peek` return `false` or `null`.	If an empty queue is an expected case, use `poll` / `peek` and check for `null`.
Taking `PriorityQueue` for FIFO	`Queue<Integer> q = new PriorityQueue<>()` returns items by priority, not by arrival.	For FIFO, use `ArrayDeque` or `LinkedList`.
`ArrayDeque` and `null`	`ArrayDeque` rejects `null` items (`NullPointerException`); a `null` item would be confused with `poll`'s "empty" result.	Never put `null` in a queue; if needed, use a separate "empty" object.

Table 8.24: Common mistakes in queue code.

The last row is a design decision: `poll()` returns `null` when empty, so a `null` item would make "empty" and "front item is `null`" indistinguishable. `ArrayDeque` forbids `null`; `LinkedList` allows it, so its `poll` results need more care.

The chapter's problems share a pattern. A queue is the right tool when the question is **"Which waiting item has waited longest?"**: the process waiting longest for the processor, the next person in the Josephus count, the smallest number whose children are not yet generated. A stack asks the opposite: "Which unresolved item was seen most recently?" If items must be handled in arrival order, think queue; if both ends matter, as in the sliding window, think deque.

8.7 Summary

- A **queue** inserts at the rear and removes at the front: **FIFO**. A good implementation makes `enqueue`, `dequeue`, `first` / `peek`, `size`, and `isEmpty` all *O(1)*.
- Removing from an empty queue is **underflow**; adding to a full fixed-capacity one, **overflow**. In Java's `Queue`, `add` / `remove` / `element` throw, while `offer` / `poll` / `peek` return `false` or `null`. `ArrayDeque` is array-based, `LinkedList` linked.

- A **simple array queue** cannot reuse slots freed at the front and can report "full" while nearly empty. Shifting fixes this but makes removal *O(n)*.
- A **circular queue** advances indices with *(i + 1) % N* and reuses slots. With a counter, the insertion point is `(front + size) % N`, empty is `size == 0`, and full is `size == N`; with one empty slot it holds at most *N − 1* items and is full when `(rear + 1) % N == front`.
- A full circular queue **grows** by copying into a double-size array in `(front + k) % N` order: one resize is *O(n)*, but *n* insertions cost under *3n*, amortized *O(1)* each. `Arrays.copyOf` scrambles a wrapped queue.
- A **linked queue** inserts at the end and removes at the head through `front` and `rear`, both in worst-case *O(1)*; the first insertion must set `front`, and removing the last item must clear `rear`.
- A **deque** inserts and removes at both ends: one end only makes it a stack, one end in and the other out a queue. In a doubly linked list, `prev` makes rear removal *O(1)*; removing the last item must clear both ends.
- **Sliding window maximum** takes *O(n)* with a deque of indices whose values decrease front to rear; each index enters once and leaves at most once.
- **Round robin** runs each process for at most *q* units and requeues unfinished ones at the rear. **Josephus** takes *O(n·k)*, moving *k − 1* people to the rear and eliminating the *k*th. **Binary numbers** come out in increasing order by queueing the children "x0" and "x1."
- A **queue from two stacks** pushes onto one stack and transfers to the other when it is empty. Each item moves at most once, so operations are amortized *O(1)*, though one `dequeue` can be *O(n)*.

Implementation	Insert	Remove	Front item	Note
Simple array (shifting)	O(1)	O(n)	O(1)	shifts on every removal
Circular array	O(1)	O(1)	O(1)	fixed capacity, % N
Linked list	O(1)	O(1)	O(1)	`front` + `rear` references
Deque (doubly linked list)	O(1) both ends	O(1) both ends	O(1)	`prev` reference
Queue from two stacks	O(1)	O(1) amortized	O(1) amortized	each item transferred once

Table 8.25: Costs of the queue implementations.

8.8 Key terms

Each term links to the course glossary, which gives its Turkish equivalent and its use in other chapters.

queue · FIFO (first-in, first-out) · front · rear · enqueue · dequeue · abstract data type (ADT) · interface · generic · overflow · underflow · array · capacity · circular queue · amortized analysis · singly linked list · doubly linked list · node · deque (double-ended queue) · priority queue · sliding window · scheduler · round robin

time slice (quantum) | breadth-first search (BFS) | stack | time complexity | space complexity

8.9 Test Yourself

1. Starting empty, a queue receives `enqueue(5)`, `enqueue(8)`, `dequeue()`, `enqueue(2)`, `first()`, `enqueue(6)`, `dequeue()`, `dequeue()`. What do the value-returning calls return, and what remains?

   Answer

   `dequeue()` returns 5, leaving 8. After `enqueue(2)` the queue is 8, 2, and `first()` returns 8 without removing it. After `enqueue(6)` it is 8, 2, 6; the two `dequeue()` calls return 8 and 2. Returned: 5, 8, 8, 2; only 6 remains.

2. A counter-based circular queue has $N = 8$, `front = 5`, `size = 6`. Where are the items, where is the last one, and where does the next insertion go?

   Answer

   The items are at $(5 + k) \% 8$ for $k = 0, \ldots, 5$: 5, 6, 7, 0, 1, 2. The last is at $(5 + 6 - 1) \% 8 = 2$, and the next insertion goes to $(5 + 6) \% 8 = 3$. Two more insertions fit (slots 3 and 4) before `size == 8`.

3. An empty counter-based circular queue of capacity 4 receives `enqueue(1)`, `enqueue(2)`, `enqueue(3)`, `dequeue()`, `dequeue()`, `enqueue(4)`, `enqueue(5)`, `enqueue(6)`. What are the array, `front`, and `size` at the end, and what does a further `enqueue(7)` do?

   Answer

   Slots 0–2 fill; the removals return 1 and 2, giving `front = 2`, `size = 1`. Then 4 goes to $(2 + 1) \% 4 = 3$, 5 to $(2 + 2) \% 4 = 0$, 6 to $(2 + 3) \% 4 = 1$. The array is `[5, 6, 3, 4]`, `front = 2`, `size = 4`: the queue 3, 4, 5, 6 is full, and `enqueue(7)` throws `IllegalStateException("full")`.

4. A full circular queue of capacity 3 has `data = [R, P, Q]` and `front = 1`. What are the array, `front`, and `size` after a growing `enqueue(S)`?

   Answer

   The logical order is P, Q, R. Since `size == 3`, `resize(6)` runs: `temp[0] = data[1] = P`, `temp[1] = data[2] = Q`, `temp[2] = data[(1 + 2) % 3] = data[0] = R`, and `front = 0`. S goes to $(0 + 3) \% 6 = 3$, giving `[P, Q, R, S, _, _]`, `front = 0`, `size = 4`.

5. Suppose `if (front == null) rear = null;` is deleted from the linked queue's `dequeue`. Trace `enqueue(7)`, `dequeue()`, `enqueue(9)` from an empty queue. When does the bug become visible?

   Answer

   After `enqueue(7)` both ends point to node 7. `dequeue()` returns 7 and nulls `front`, but `rear` stays on node 7 with `size = 0`. `enqueue(9)` finds the queue empty and moves both `front` and (via `rear = temp`) `rear` to 9, so the final state is correct. The bug shows only if `rear` is read while the queue is empty: `rear()` would return 7, which is not in the queue, and node 7 cannot be freed.

6. On `Deque<Integer> d = new ArrayDeque<>();` we call `d.offerLast(1)`, `d.offerFirst(2)`, `d.push(3)`, `d.offer(4)`, `d.pollLast()`, `d.pop()`. What do the last two calls return, and what does printing `d` show?

   Answer

   `offerLast(1)` → [1]; `offerFirst(2)` → [2, 1]; `push(3)` adds at the front → [3, 2, 1]; `offer(4)` adds at the rear → [3, 2, 1, 4]. `pollLast()` returns 4 → [3, 2, 1]; `pop()` returns 3 from the front → [2, 1]. `System.out.println(d)` prints `[2, 1]`.

7. What does `windowMax({2, 7, 3, 1, 5, 4}, 2)` return, and how many times is `pollLast()` called?

   Answer

   The result is `[7, 7, 3, 5, 5]`. At *i = 1*, 7 drops 0:2 from the rear (1st `pollLast`); at *i = 2*, 3 is added; at *i = 3*, 1:7 leaves from the front and 1 is added; at *i = 4*, 2:3 leaves from the front and 5 drops 3:1 (2nd `pollLast`); at *i = 5*, 4 is added. Two `pollLast` and two `pollFirst` calls in all, fewer than the 6 insertions.

8. The ready queue holds P1 (3 units), P2 (4 units), and P3 (1 unit), and *q = 2*. When does each process finish under round robin, and what is the average waiting time?

   Answer

   0–2: P1 (1 left), queue P2, P3, P1. 2–4: P2 (2 left), queue P3, P1, P2. 4–5: P3 finishes. 5–6: P1 finishes. 6–8: P2 finishes. Completion: P3 = 5, P1 = 6, P2 = 8. Waiting times *6 − 3 = 3*, *8 − 4 = 4*, *5 − 1 = 4*; average *11/3 ≈ 3.67* units.

9. For six people numbered 1 to 6, in what order does `josephus(q, 3)` eliminate them, and whom

does it return? Check the result with $J(n) = (J(n - 1) + k) \% n$.

> **Answer**
>
> The order is 3, 6, 4, 2, 5, and 1 survives. Numbering from 0: $J(1) = 0$, $J(2) = 1$, $J(3) = 1$, $J(4) = 0$, $J(5) = 3$, $J(6) = (3 + 3) \% 6 = 0$; position 0 is person 1, so the methods agree. The queue performs $5 \cdot 3 = 15$ operations.

10. A two-stack queue receives `enqueue(a)`, `enqueue(b)`, `dequeue()`, `enqueue(c)`, `enqueue(d)`, `dequeue()`, `dequeue()`, `dequeue()`. What do the removals return, which ones transfer, and how many items move in total?

> **Answer**
>
> 1st `dequeue`: `outbox` is empty, so b and a move (2 items) and a is returned. c and d go into `inbox`. 2nd: `outbox` holds b, no transfer; b is returned. 3rd: `outbox` is empty, d and c move (2 items), and c is returned. 4th: d. Output a, b, c, d; transfers on the 1st and 3rd calls, 4 items in all.

8.10 Further reading

- M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, *Data Structures and Algorithms in Java* (6th ed., 2014), Sections 6.2 and 6.3: the queue ADT, circular-array and linked implementations, and the deque, with a Java interface close to ours.
- R. Sedgewick and K. Wayne, *Algorithms* (4th ed., 2011), Section 1.3: linked and resizing-array queues in short, clear code, with iterators and generics.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms* (3rd ed., 2009), Section 10.1: a formal array implementation of the queue; the exercises include the two-stack queue and the deque.
- A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts* (10th ed., 2018), Chapter 5: CPU scheduling, including round robin, the choice of quantum, and criteria such as waiting time.

8.11 References

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 3rd ed., MIT Press, 2009.
2. M. T. Goodrich, R. Tamassia, M. H. Goldwasser, *Data Structures and Algorithms in Java*, 6th ed., Wiley, 2014.
3. R. Lafore, *Data Structures and Algorithms in Java*, 2nd ed., Sams, 2002.
4. Oracle, *Java SE 17 API Specification*: documentation for `java.util.Queue`, `java.util.Deque`, `java.util.ArrayDeque`, and `java.util.PriorityQueue`.
5. R. Sedgewick, K. Wayne, *Algorithms*, 4th ed., Addison-Wesley, 2011.

6. A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10th ed., Wiley, 2018.

PART III

Trees

Chapters 9–11

Trees

Arrays, linked lists, stacks, and queues keep their elements in a line: each has at most one predecessor and one successor. Much real data is hierarchical instead — folders inside folders, chapters inside a book, elements nested in a web page — and the structure that holds it naturally is the tree. We start with its vocabulary (root, parent, child, leaf, degree, depth, height), then turn to binary trees, their full, complete, and perfect varieties, and how node count bounds height. After storing trees with arrays and with links, we reach the heart of the chapter: preorder, inorder, postorder, and level-order traversals, coded recursively and then with an explicit stack and queue, each traced step by step. The last sections cover expression trees, recursive tree methods, and rebuilding a tree from two traversals.

Code for this chapter: `github.com/sercankulcu/data-structures-java/Bolum09/src`

LEARNING OUTCOMES

Core concepts
Explain root, parent, child, sibling, ancestor, descendant, leaf, degree, depth, and height, and find these values in a given tree.

Representations
Represent trees with linked nodes and with arrays; compare the memory and access costs of the two.

Traversals
Carry out preorder, inorder, postorder, and level-order traversals by hand and in code; trace recursive calls on the call stack.

Applications and cost
Compare general and binary trees by their uses and operation costs; write recursive methods that work on trees.

9.1 The tree concept

Every structure so far has been **linear**: each element is followed by at most one other. An organization chart, a file system, or a family tree will not fit on a line, because an element can have several elements below it, each with its own below it. This section sets up the language of trees, the structure for such hierarchies.

9.1.1 What is a tree?

A **tree** is a nonlinear structure that stores elements hierarchically. Each element is a **node**, and a line joining two nodes is an **edge**. The single node at the top is the **root**; computer scientists draw trees upside down, as in Figure 9.1.

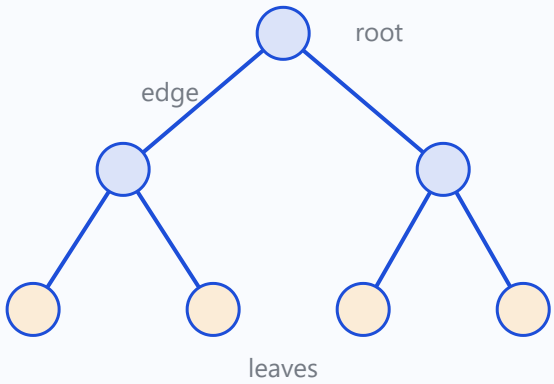


Figure 9.1: A tree with seven nodes. The root at the top branches into two children, which branch into their own. The four bottom nodes have no children: they are leaves. Six edges join the seven nodes.

Each edge joins an upper node, the **parent**, to a lower one, its **child**. One rule makes a tree a tree: every node except the root has exactly one parent, and the root has none. The number of children is unlimited — zero, one, or many.

Definition 9.1

A **tree** is a structure of nodes and of edges joining them in parent–child relationships. A nonempty tree satisfies two conditions: (1) exactly one node, the **root**, has no parent; (2) every other node has exactly one parent, and from every node we reach the root by climbing from parent to parent.

Recursively: a tree is either empty or a root with zero or more trees hanging below it, the root's **subtrees**.

The second condition rules out a **cycle**: you cannot leave a node, walk along edges without reusing one, and return. A cycle would force some node on it either to have two parents or to climb forever without reaching the root.

The most used numerical fact about trees concerns edges. Every non-root node is joined to its parent by exactly one edge, so pairing each edge with its lower end is a one-to-one correspondence.

Theorem 9.1 (Number of edges)

Every tree with $n \geq 1$ nodes has exactly $n - 1$ edges.

Proof. Pair each edge with its lower end, the child. A node has one parent, so each child gets at most one edge; the root is the lower end of no edge, and every other node of exactly one. The edges thus match the $n - 1$ non-root nodes. □

Figure 9.1 agrees: seven nodes, six edges. The rule works both ways: the node count gives the edge count at once, and a connected structure with at least as many edges as nodes must contain a cycle.

9.1.2 Hierarchies in everyday life

People drew hierarchies as trees long before computers.

- **File system.** The root directory (`/` on Unix-like systems) is at the top. Folders hold folders and files; files hold nothing. Folders are internal nodes, files leaves.
- **Book.** Parts divide into chapters, chapters into sections; the table of contents is the tree written with indentation.
- **Geography.** Continents contain countries, countries provinces, provinces cities.
- **Web page.** `<html>` contains `<head>` and `<body>`, which contain further elements; the browser builds a tree of them, the **Document Object Model** (DOM).

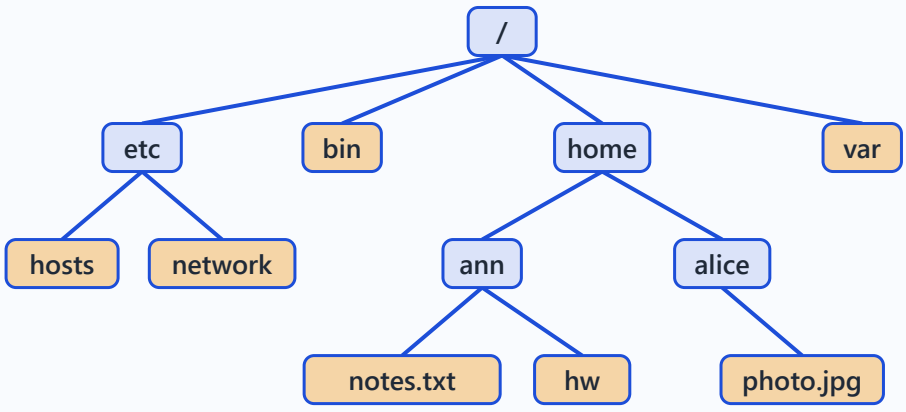


Figure 9.2: A file system tree. The root directory `/` holds `etc`, `bin`, `home`, and `var`; `home` holds the folders

`ann` and `alice`. The orange boxes are files, the leaves of the tree.

Figure 9.2 also shows why a file has one full path: `notes.txt` is reached only along `/home/ann/notes.txt`. With one parent per node, the path down from the root is unique. Shortcut (link) files add extra connections, and the structure stops being a pure tree.

9.1.3 When is a structure a tree?

Without a marked root there is no notion of parent, so for an arbitrary linked structure we restate the definition: the structure must be **connected** (a path joins every pair of nodes) and have no **cycle**. Then it has one edge fewer than nodes, and any node can be chosen as the root; each node's parent is then its unique neighbor toward the root.

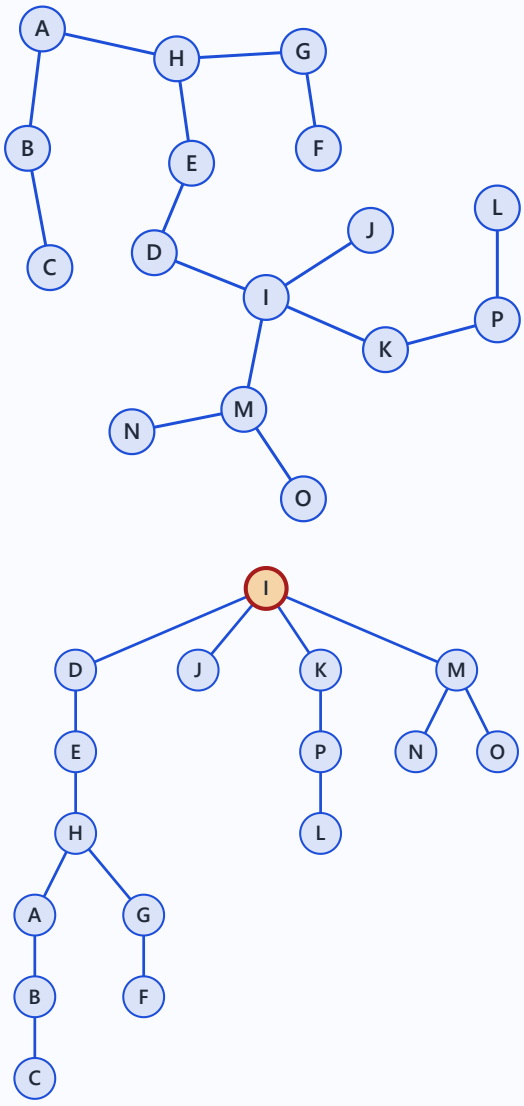


Figure 9.3: Left: a linked structure with 16 nodes, A to P, and no root marked. Right: the same structure redrawn with I as the root. I's children are D, J, K, and M; A, B, and C hang below D–E–H.

Example 9.1

Is the structure on the left of Figure 9.3 a tree?

1. **Nodes and edges.** 16 nodes, A to P, and 15 edges: A–B, B–C, A–H, H–G, G–F, H–E, E–D, D–I, I–J, I–K, K–P, P–L, I–M, M–N, M–O.
2. **Connectivity.** From I we reach D, J, K, M; from D, E; from E, H; from H, A and G; from A, B; from B, C; from G, F; from K, P; from P, L; from M, N and O — all 16 nodes.
3. **Conclusion.** Connected, with 15 edges for 16 nodes: no cycle, so a tree. Any node, I for instance, can be the root, as drawn on the right.

We never checked for cycles; the count did it. Connecting n nodes takes at least $n - 1$ edges, and one more edge joins two nodes that already have a path between them, closing a cycle. So "connected with $n - 1$ edges" means "connected and acyclic."

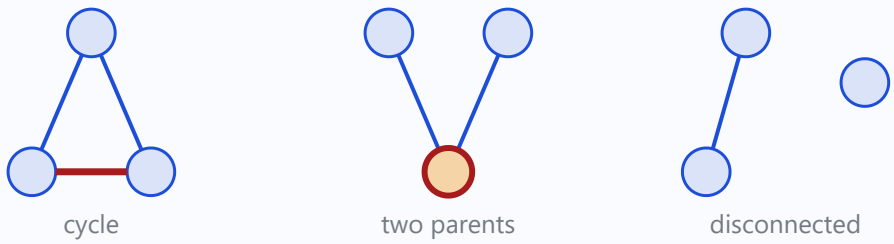


Figure 9.4: Three non-trees. Left: three nodes form a cycle (the red edge closes it). Middle: the bottom node has two parents. Right: two disconnected pieces.

Figure 9.4 shows the three typical violations. The triangle has three edges for three nodes, one too many, so a cycle forms. In the middle, whatever the root, the bottom node is reached by two paths. On the right, three nodes share one edge; with an edge missing, the structure splits. A collection of disjoint trees like this is a **forest**.

Common mistake

Calling a structure a tree because it is connected. A connected structure with 10 nodes and 10 edges is not a tree: a tree has 9, and the extra edge closes a cycle. Nor are $n - 1$ edges enough alone: a piece with a cycle plus an isolated node can have them too. Check both connectivity and $n - 1$ edges (or connectivity and no cycle).

9.1.4 Root, internal nodes, and leaves

Whether a node has children matters in nearly every tree algorithm: a recursive method usually stops at a leaf and descends into the children of an internal node.

- **Root**: the top node, the one without a parent — the root directory of a file system.
- **Internal node**: a node with at least one child — a folder.
- **Leaf**: a node with no children, also called an **external node** — a file.

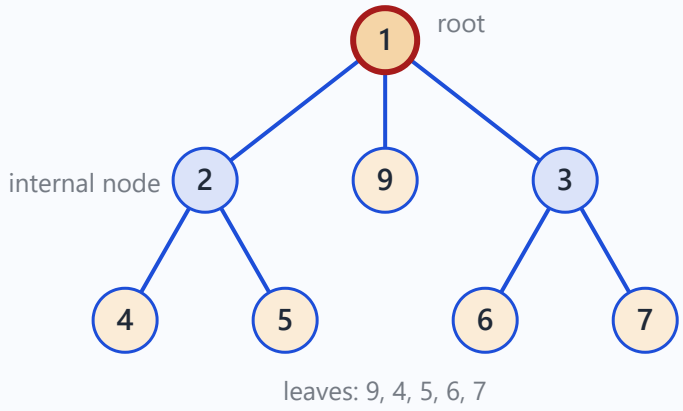


Figure 9.5: A tree rooted at 1. Nodes 2 and 3 are internal; 9, 4, 5, 6, and 7 have no children and are leaves. The root has children, so it is internal too.

Figure 9.5 makes two subtle points. First, “root” and “internal node” overlap: the root is defined by position, internal nodes and leaves by their number of children. Second, leaves need not share a level: 9 is a child of the root and a leaf, like 4, 5, 6, and 7. In a one-node tree the root is a leaf.

Common mistake

Looking for leaves only in the bottom row. A leaf is a node without children, wherever it is drawn: node 9 in Figure 9.5 sits in the second row and is a leaf. Nor should the root always be ruled out: in a one-node tree it is a leaf.

9.1.5 Relationships between nodes

The terminology comes from family trees. A node's **parent** is directly above it, its **children** directly below, and children of one parent are **siblings**. The nodes on the path up to the root are its **ancestors**; all nodes reachable going down are its **descendants**. Table 9.1 illustrates these on Figure 9.6.

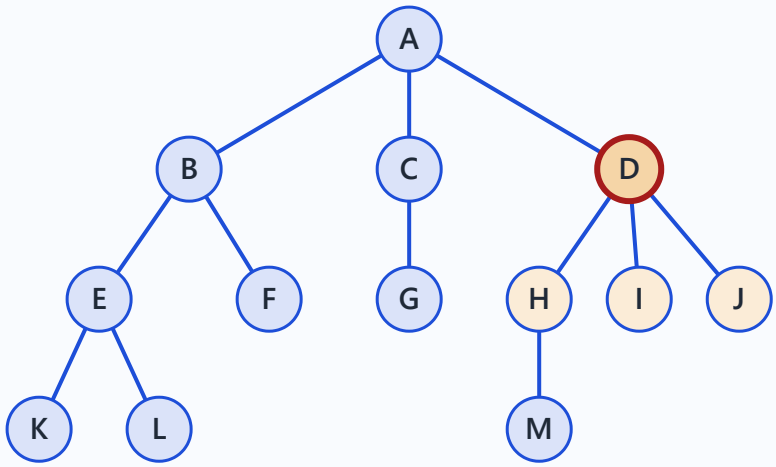


Figure 9.6: An example tree. A is the root, with children B, C, and D; D's children H, I, J are highlighted. M is below H, E and F below B, K and L below E, G below C.

Term	Meaning	Example
Parent	The node directly above	D is H's parent.
Child	A node directly below	H, I, and J are D's children.
Sibling	A node with the same parent	H, I, and J are siblings.
Ancestor	A node on the path to the root	M's ancestors are H, D, and A.
Descendant	A node in the subtree	D's descendants are H, I, J, and M.

Table 9.1: Relationships in a tree, with examples from Figure 9.6.

Ancestor and descendant are inverses: M's ancestors are H, D, A, so M is a descendant of each. The root is everyone's ancestor, a leaf no one's. Some sources count a node as its own ancestor and descendant; here they exclude the node, so a node's descendants are the other nodes of its **subtree**.

Sharing a row does not make nodes siblings. In Figure 9.6, F's parent is B and G's is C, so they are cousins, not siblings; siblinghood means a shared parent.

Common mistake

Treating nodes on one level as siblings, or a node's descendants as just its children. D's descendants are the whole subtree: H, I, J, and also H's child M. Likewise M's ancestors are the whole path to the root, H, D, A, not just its parent.

Figure 9.7 shows the tree in four frames of the slide drawing that introduces the terms. Of its 13 nodes, A through M, seven are leaves (K, L, F, G, M, I, J) and six internal (A, B, C, D, E, H); 7 + 6 = 13, and by Theorem 9.1 there are 12 edges.

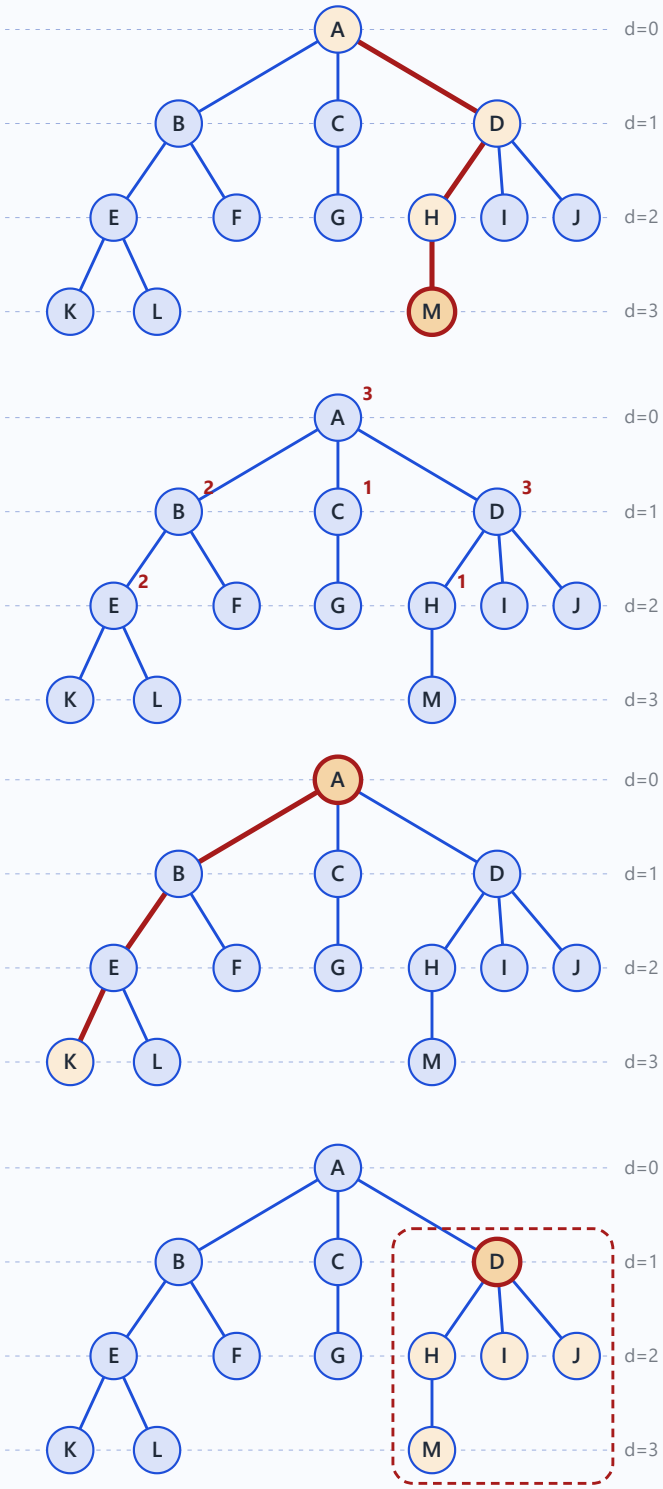


Figure 9.7: Terms on the 13-node tree (left to right, top to bottom). (a) M's ancestors: the path M–H–D–A. (b) Degrees: A 3, B 2, C 1, D 3, E 2, H 1; the tree has degree 3. (c) Height: the path A–B–E–K to the farthest leaf, 3 edges. (d) D's subtree: D, H, I, J, M. Dashed lines mark depths d = 0, 1, 2, 3.

9.1.6 Degree, depth, level, and height

Nearly every cost analysis on trees rests on a few measures of how much a node branches and how far it is from the root.

Definition 9.2

- The **degree** of a node is its number of children (subtrees); a tree's degree is the largest degree of its nodes.
- The **depth** of a node is the number of edges on the path from the root to it; the root has depth 0.
- The nodes at depth d form **level** d.
- The **height** of a node is the number of edges on the path from it down to its farthest leaf; a leaf has height 0.
- The **height of a tree** is the height of its root, which equals the largest depth.

Depth looks up: how far below the root is this node? Height looks down: how long is the longest branch beneath it? The root has depth 0 and the tree's height; a leaf has height 0 and a depth that depends on where it sits.

Measure	Definition	Example
Degree	Number of children of the node	D → 3
Depth	Edges from the root to the node; 0 at the root	E → 2 (A→B→E)
Level	The nodes at the same depth	Level 2: E, F, G, H, I, J
Node height	Edges from the node to the farthest leaf; 0 at a leaf	D → 2 (D→H→M)
Tree height	Height of the root = largest depth	3

Table 9.2: Tree measures and their values in the 13-node tree of Figure 9.7.

These notes count **edges**. Some books count **nodes** and number levels from 1, so every depth and height comes out one larger. Both are consistent, but mixing them shifts results by one; check a source's convention before comparing.

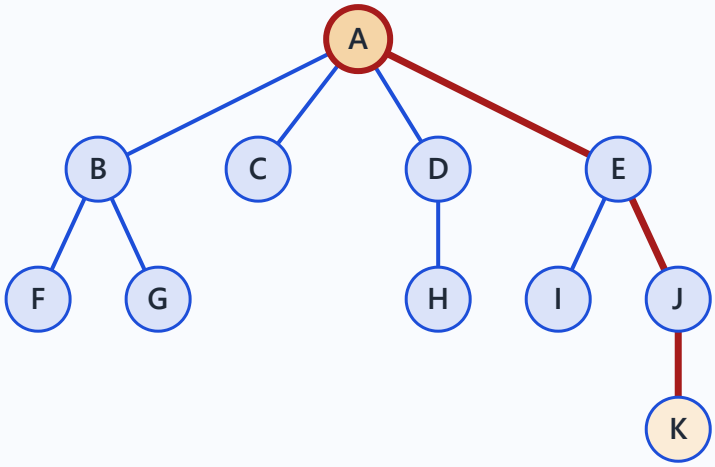


Figure 9.8: A tree for depth and height. A's children are B, C, D, E; B's are F and G; D's is H; E's are I and J; J's is K. The highlighted path A→E→J→K is the longest.

Example 9.2

Find the depth and height of every node in Figure 9.8.

1. **Depths, top down.** A has depth 0; B, C, D, E depth 1; F, G, H, I, J depth 2; K depth 3.
2. **Heights, bottom up.** Leaves C, F, G, H, I, K have height 0; an internal node's height is one more than its children's largest. J: *1 + 0 = 1*. B: *1 + max(0, 0) = 1*. D: 1. E, with children I (0) and J (1): *1 + max(0, 1) = 2*.
3. **The root.** A's children have heights 1, 0, 1, 2, so A's height is *1 + 2 = 3* — the tree's height, equal to the depth of the deepest node, K.

Node	A	B	C	D	E	F	G	H	I	J	K
Height	3	1	0	1	2	0	0	0	0	1	0
Depth	0	1	1	1	1	2	2	2	2	2	3

Table 9.3: Height and depth in the tree of Figure 9.8.

Depth depends only on a node's ancestors, height on all its descendants. The code mirrors this: depth climbs the parent chain, height descends through the children.

Common mistake

Computing a node's height as the tree's height minus its depth. That works only on the longest branch. In Figure 9.8, B has depth 1 and the tree height 3, yet B's height is 1, not *3 − 1 = 2*: the longest branch below B is one edge. Height always comes from the node's own subtree.

9.1.7 Paths and subtrees

Walking from node to node along edges traces a **path**, formally a sequence alternating nodes and edges:

> $p = (v_0, e_1, v_1, e_2, \ldots, e_k, v_k)$

Each edge e_i joins v_{i-1} and v_i; the path has $start(p) = v_0$, $end(p) = v_k$, and length $length(p) = k$, its number of edges. A path repeating no node is a **simple path**. Depth and height are lengths of paths from the root down to a node and from a node down to its farthest leaf.

Theorem 9.2 (Unique path)

In a tree, there is exactly one simple path between any two nodes.

Proof. Existence: climb the parent chains of both nodes toward the root, and let c be the first node common to both. Climbing from the first node to c and descending to the second gives a simple path. Uniqueness: two different simple paths would split somewhere and later rejoin, and going out along one and back along the other would form a cycle, which a tree does not have. □

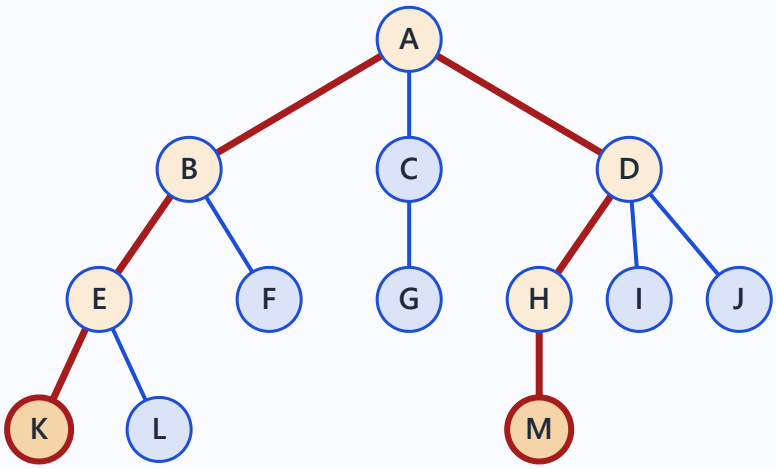


Figure 9.9: The unique path K → E → B → A → D → H → M. It climbs to A, the first common ancestor, then descends to M: 6 edges.

Figure 9.9 makes the proof concrete. K's ancestors are E, B, A and M's are H, D, A; the chains first meet at A, so the path climbs three edges and descends three, length 6, and no other path exists. “A file has exactly one full path” is a special case.

A **subtree** is a node together with all its descendants, a tree in its own right. The dashed frame in panel (d) of Figure 9.7 marks D's subtree: D, H, I, J, M. A folder with everything under it is a subtree, so copying or deleting a folder copies or deletes a subtree.

Subtrees make tree algorithms naturally recursive. A node's children are the roots of its subtrees, so a job on a tree can usually be phrased as: do it on each subtree, combine the results, and account for the root. Height is the first example: one more than the largest subtree height.

9.1.8 Basic kinds of trees

The definition puts no limit on children, but limiting or ordering them often helps. Table 9.4 lists the four kinds we meet in this course.

Kind	Property
General tree	No limit on the number of children.
***k*-ary tree**	Every node has at most k children.
Binary tree	Every node has at most two children: left and right.
Binary search tree (BST)	A binary tree in which, at every node, keys in the left subtree are smaller than the node's key and keys in the right subtree are larger.

Table 9.4: Basic kinds of trees.

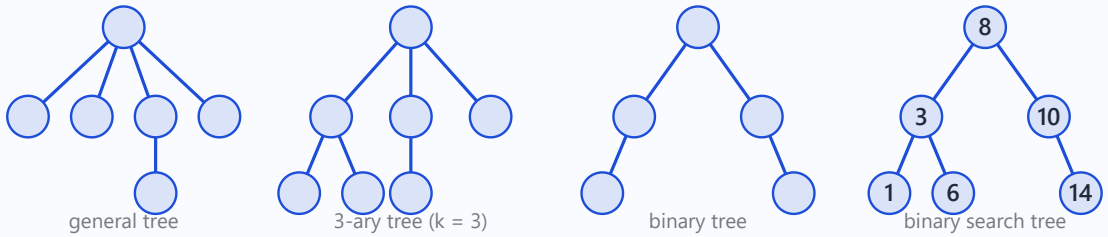


Figure 9.10: Four kinds (left to right): a general tree, a tree with at most three children ($k = 3$), a binary tree, and a binary search tree with root 8, left child 3 (children 1 and 6), and right child 10 (right child 14).

The kinds nest: every BST is a binary tree, every binary tree a k-ary tree with $k = 2$, every k-ary tree a general tree. More constraints mean less flexibility but easier work. A BST search descends into one child per node, left for a smaller key, right for a larger: finding 6 on the right of Figure 9.10 goes left from 8 and right from 3, two comparisons.

These notes concentrate on binary trees. Every general tree can be stored as a binary tree, as the section on representations shows. Insertion, deletion, and balancing in BSTs are a large topic of their own, mentioned here only by definition.

9.1.9 The tree ADT

Before using a tree in code, we must decide which operations give access to it. As with stacks and queues, it pays to separate the operations from their implementation; the description of what a structure does is its **abstract data type** (ADT). The Java interface below is the tree ADT of Goodrich, Tamassia, and Goldwasser.

```
public interface Tree<E> extends Iterable<E> {
  Position<E> root();
  Position<E> parent(Position<E> p);
  Iterable<Position<E>> children(Position<E> p);
  int numChildren(Position<E> p);
  boolean isInternal(Position<E> p);  // has a child
  boolean isExternal(Position<E> p);  // leaf
  boolean isRoot(Position<E> p);      // no parent
  int size();
  boolean isEmpty();
  Iterable<Position<E>> positions(); // all positions
}
// Position<E>: E getElement();
```

The interface is built around **positions**. A `Position<E>` stands for a place in the tree and has one operation, `getElement()`, returning the element there. The user never sees, references, or modifies the node objects; all access goes through ADT methods on positions. Whether the tree is linked or stored in an array is hidden, so client code survives a change of implementation.

The methods fall into three groups.

- **Navigation.** `root()` returns the root's position (`null` if empty); `parent(p)` returns `p`'s parent, or `null` at the root; `children(p)` yields the children in order, as an `Iterable` usable in a `for-each` loop; `numChildren(p)` is `p`'s degree.
- **Queries.** `isInternal(p)`, `isExternal(p)`, and `isRoot(p)` report whether a position is internal, a leaf, or the root. The names follow the reference textbook.
- **General methods.** `size()` and `isEmpty()` report the node count and emptiness; `positions()` iterates over all positions and `iterator()` (from `Iterable<E>`) over all elements.

The binary tree ADT adds three methods: `left(p)` and `right(p)` return the children (or `null`), and `sibling(p)` the sibling.

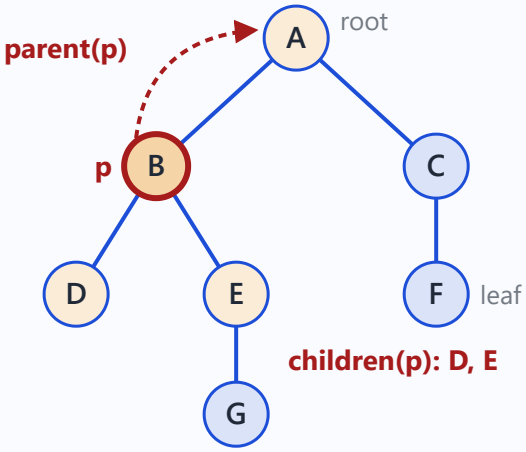


Figure 9.11: The ADT methods on an example. `p` is the position of node B. `parent(p)` gives the root A, `children(p)` gives D and E. D is a leaf and A the root; E's only child is G, C's only child F.

On Figure 9.11: `numChildren(p)` returns 2; `isInternal(p)` is `true`, `isExternal(p)` and `isRoot(p)` `false`; `root()` gives A, `parent(root())` is `null`, and `size()` is 7.

9.1.10 Code trace: depth and height

With the ADT we can compute depth and height without knowing the implementation; both methods transcribe their definitions almost word for word.

```
int depth(Position<E> p) {
    if (isRoot(p)) return 0;
    return 1 + depth(parent(p));
}
int height(Position<E> p) {
    int h = 0;
    for (Position<E> c : children(p))
        h = Math.max(h, 1 + height(c));
    return h;
}
```

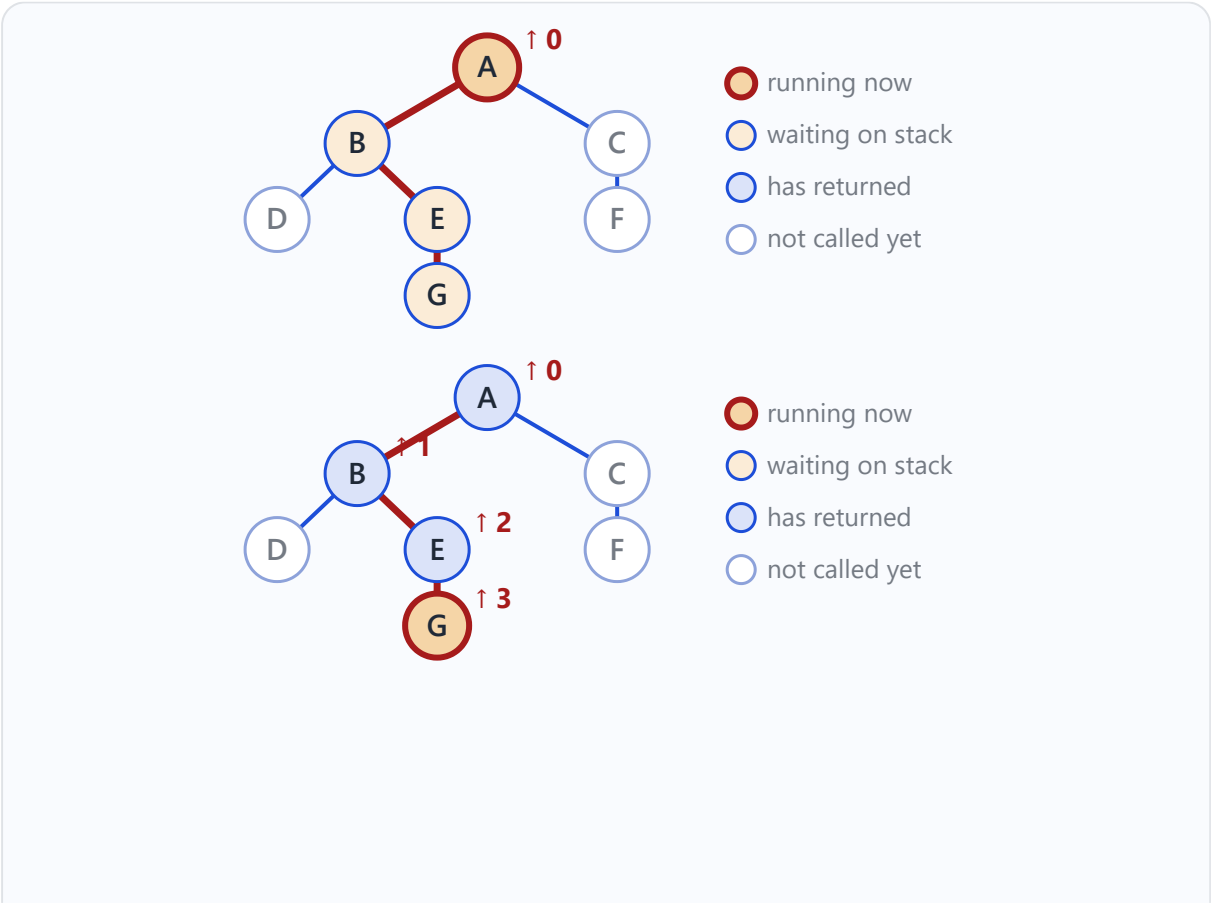
`depth(p)` follows the definition: 0 at the root, otherwise one more than the parent's depth. It climbs the parent chain one edge per call, so for depth d it is called $d + 1$ times: $O(d + 1)$ time and at most $d + 1$ stack frames.

`height(p)` looks downward. Starting from `h = 0`, the loop raises `h` to `1 + height(c)` for each child `c`; at a leaf the loop never runs and 0 is returned. The method is called once per node of `p`'s subtree, and iterating over children costs their number, which sums to the edge count, so a subtree of n_p nodes takes $O(n_p)$.

Let us trace `depth(G)` and then `height(B)` on Figure 9.11. Table 9.5 gives the call stack (bottom to top) and returned values.

Step	Running call	Call stack	What happens
1	`depth(G)`	main, depth(G)	G is not the root; `depth(E)` is called for its parent E.
2	`depth(E)`	..., depth(E)	E is not the root; `depth(B)` is called.
3	`depth(B)`	..., depth(B)	B is not the root; `depth(A)` is called.
4	`depth(A)`	main, G, E, B, A (5 frames)	A is the root: base case, returns 0.
5	`depth(B)`	main, G, E, B	Returns *1 + 0 = 1*.
6	`depth(E)`	main, G, E	Returns *1 + 1 = 2*.
7	`depth(G)`	main, G	Returns *1 + 2 = 3*. G has depth 3 (A–B–E–G).
8	`height(B)`	main, height(B)	`h = 0`; children D, E. Recurse on the first child, D.
9	`height(D)`	..., B, D	D has no children; the loop does not run, returns 0.
10	`height(B)`	main, B	`h = max(0, 1 + 0) = 1`; next child E.
11	`height(E)`	..., B, E	`h = 0`; recurse on the only child, G.
12	`height(G)`	..., B, E, G	G is a leaf: returns 0.
13	`height(E)`	..., B, E	`h = max(0, 1 + 0) = 1`; no children left, returns 1.
14	`height(B)`	main, B	`h = max(1, 1 + 1) = 2`; no children left, returns 2.

Table 9.5: Trace of `depth(G)` and `height(B)`. The call stack is written bottom to top.



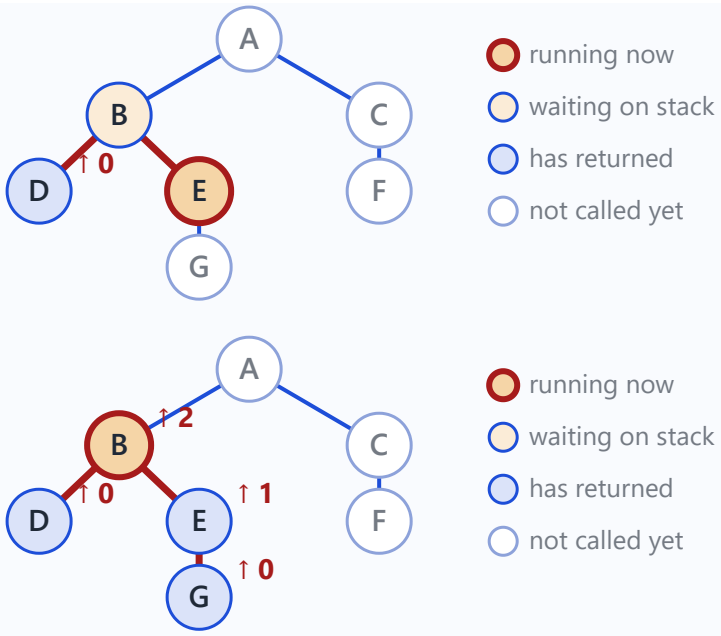


Figure 9.12: Four moments of the trace. (a) `depth(A)` is at the base case; frames for G, E, B wait. (b) Values come back: B 1, E 2, G 3. (c) `height(B)` got 0 from D and entered E's subtree. (d) G returned 0, E 1; `height(B) = 2`.

The trace shows a symmetry. `depth` calls on the way up and computes on the way down: the root says 0, and each frame adds 1. `height` calls on the way down and computes on the way up: leaves say 0, and each internal node adds 1 to its children's maximum. `height(B)` was called four times, for exactly B's subtree (B, D, E, G); A, C, F were never touched.

A tree's height can be found with `height(root())` in $O(n)$, or as the largest `depth`, which climbs separately from every node at cost $O(n + \Sigma\, d_p)$. In a chain the depths 0, 1, …, $n - 1$ sum to $n(n - 1)/2$, so that costs $O(n^2)$. Use the first method.

> **Common mistake**
>
> Starting `h` at 1 in `height`, or returning 1 at a leaf: height is then measured in nodes and every value is one too large. With the edge-based definition, a leaf returns 0. Likewise, calling `parent(p)` in `depth` without the `isRoot(p)` check carries on at the root with a `null` position and fails.

9.2 Binary trees

In a general tree a node may have any number of children, so each node must keep a variable-length child list. Limiting nodes to two children simplifies both storage and algorithms, which is why binary trees are the most widely used kind; expression trees, binary search trees, heaps, and Huffman trees are all binary.

9.2.1 Definition of a binary tree

In a binary tree every node has at most two children, but what really sets it apart is that the children are *named*: each is a **left child** or a **right child**. Even an only child is on one particular side, and the side matters.

> **Definition 9.3**
>
> A **binary tree** is either empty or a root together with two binary trees, its **left subtree** and **right subtree**. The root of a nonempty left subtree is the main root's left child; likewise on the right.

The definition is recursive but not circular: the subtrees shrink at every step until we reach the empty tree, the **base case**. It also dictates how binary tree methods are written: return a value for the empty tree; otherwise apply the method to both subtrees and combine the results.

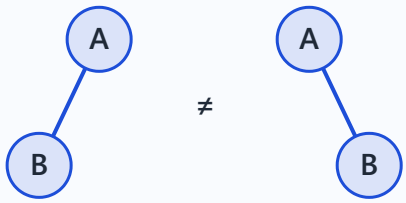


Figure 9.13: Two different binary trees: B is A's left child on the left, its right child on the right. As general trees they are the same.

In both drawings of Figure 9.13, A has one child, B. As general trees they coincide; as binary trees one has an empty right subtree and the other an empty left one. The difference matters: in a BST left means smaller keys, and inorder traversal outputs B A for the left tree but A B for the right.

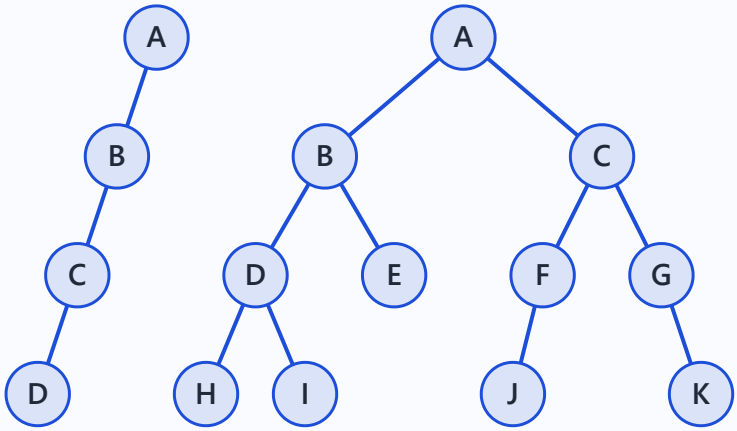


Figure 9.14: Two binary trees: a left-skewed chain A, B, C, D, and an 11-node tree with evenly spread branches.

Nodes need not be spread evenly. In the left structure of Figure 9.14 every node has only a left child, and the tree has degenerated into a chain, a **skewed** tree. It is a valid binary tree but loses most of the advantages: four nodes in a chain have height 3, evenly arranged only 2. With a million nodes, the gap becomes about 20 versus a million.

9.2.2 Full, complete, and perfect binary trees

Three names describe the shape of a binary tree. They sound alike, they are often confused, and not every source uses them the same way, so hold on to the precise definitions below.

Definition 9.4

- **Full binary tree**: every node has 0 or 2 children.
- **Complete binary tree**: every level except possibly the last is filled, and the last level is packed from the left with no gaps.
- **Perfect binary tree**: every internal node has two children and all leaves are at the same depth.

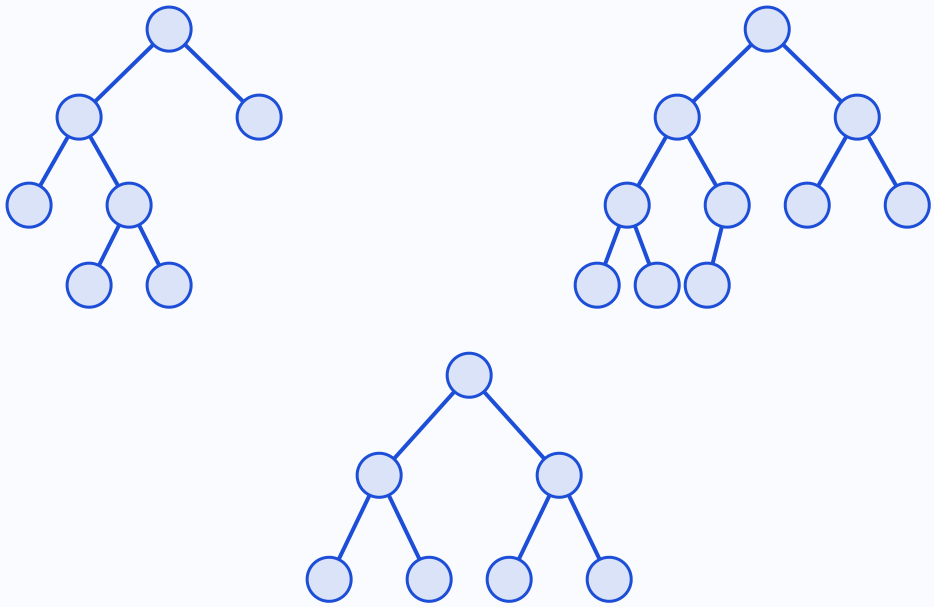


Figure 9.15: Left to right: a full binary tree (0 or 2 children everywhere), a complete binary tree (upper levels filled, the last filling from the left), and a perfect binary tree (all leaves at one depth).

The definitions look at different things. *Full* is a local condition on each node and ignores overall shape, so a full tree can be very lopsided: if the right child is always a leaf and the left one branches again, the tree is full yet nearly a chain. *Complete* is a condition on levels; it allows one node with a single child, in the right position just above the last level. *Perfect* is the strongest: the tree has every node its height allows.

To test completeness, number the nodes level by level, left to right: root 1, its children 2 and 3, then 4 to 7. A tree with n nodes is complete exactly when they fill 1 to n without gaps; if 10 is empty while 11 is used, it is not. This numbering also underlies the array representation.

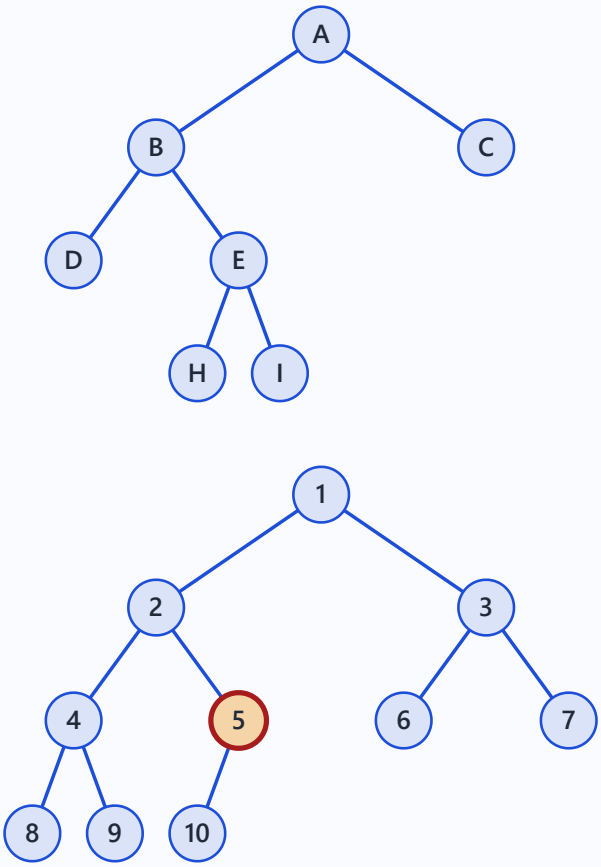
A perfect tree is both full and complete. Full and complete are otherwise independent: trees exist that are full but not complete, complete but not full, and neither.

Common mistake

Treating “full” and “complete” as the same, or assuming one implies the other. Full means 0 or 2 children everywhere; complete means all levels but the last are filled and the last fills from the left. A complete tree may have a node with one child, and a full tree may have gaps on upper levels. Check the two separately.

9.2.3 Telling them apart on four trees

We apply the three definitions to the four trees of Figure 9.16.


A
B
C
D
E
H
I
1
2
3
4
5
6
7
8
9
10

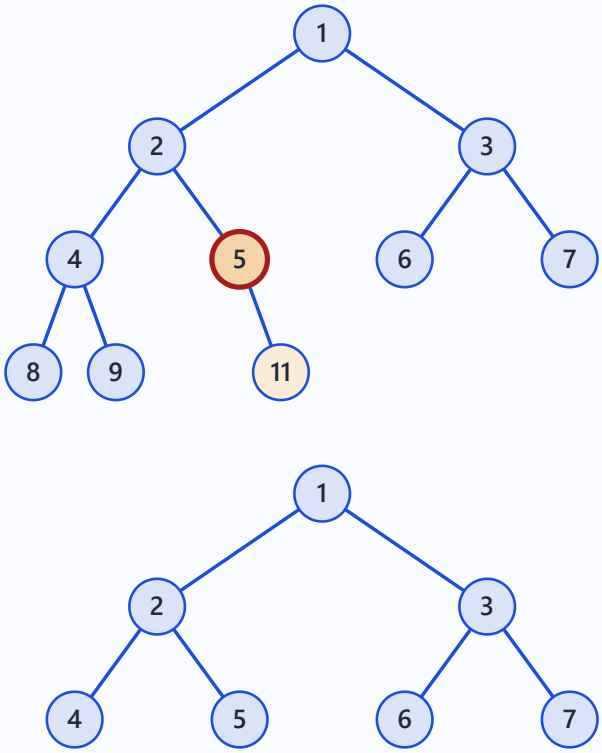


Figure 9.16: Four binary trees. (a) Root A; B's children D and E; E's children H and I; C a leaf. (b) Numbered 1 to 10, with 8, 9, 10 on the last level from the left. (c) Nodes 1 to 9 and 11; position 10 empty. (d) A perfect tree numbered 1 to 7.

Example 9.3

Is each tree in Figure 9.16 full, complete, perfect?

1. **(a)** A, B, E have 2 children and C, D, H, I none, so the tree is *full*. Level 2 holds only D and E — the positions under C are empty — yet H and I sit on level 3, so it is *not complete*. Its leaves lie at depths 1, 2, and 3, so it is *not perfect*.
2. **(b)** Numbered 1 to 10 without gaps, levels 0–2 filled and 8, 9, 10 packed left: *complete*. But 5 has one child (10), so it is *not full*, hence not perfect.
3. **(c)** Node 5 has only a right child, 11: *not full*. Position 10 is empty while 11 is used: *not complete*. So not perfect either.
4. **(d)** Internal nodes 1, 2, 3 have two children and leaves 4–7 share a depth: *perfect*, hence full and complete.

Tree	Full	Complete	Perfect
(a)	yes	no	no

Tree	Full	Complete	Perfect
(b)	no	yes	no
(c)	no	no	no
(d)	yes	yes	yes

Table 9.6: Classifying the four trees of Figure 9.16.

Rows (a) and (b) prove that full and complete are independent. Compare (b) and (c): in both, 5 has one child, but on the left (10) the numbering continues without gaps, while on the right (11) it leaves 10 empty. One node on the wrong side makes the tree incomplete.

9.2.4 Node count and height

With at most two children per node, each level holds at most twice as many nodes as the one above. Every relation between node count and height follows from this; Table 9.7 collects them, and the justifications follow.

Property	Formula	Example
Max nodes at depth *d*	2^d	$d = 3 \to$ at most 8 nodes
Max nodes in a tree of height *h*	$2^{h+1} - 1$	$h = 3 \to 15$ (perfect tree)
Min nodes in a tree of height *h*	$h + 1$	skewed tree (chain)
Height of a tree with *n* nodes	$\lfloor \log_2 n \rfloor \le h \le n - 1$	$n = 15 \to 3 \le h \le 14$
Edges in a tree with *n* nodes	$n - 1$	each node has one parent
Leaves in a full binary tree	internal nodes + 1	7 nodes → 3 internal, 4 leaves

Table 9.7: Properties of binary trees.

Theorem 9.3 (Bounds on node count)

In a binary tree, (a) there are at most 2^d nodes at depth d; (b) a tree of height h has n nodes with $h + 1 \le n \le 2^{h+1} - 1$; (c) a tree with $n \ge 1$ nodes has height in the range $\lfloor \log_2 n \rfloor \le h \le n - 1$.

Proof. (a) By induction: depth 0 holds only the root, $2^0 = 1$; if depth d holds at most 2^d nodes, each with at most two children, depth $d + 1$ holds at most 2^{d+1}.

(b) Upper bound: depths run from 0 to h, so by (a) there are at most $1 + 2 + \ldots + 2^h = 2^{h+1} - 1$ nodes. Lower bound: a path of h edges from the root to the deepest leaf contains $h + 1$ nodes.

(c) Solving (b) for h: $h + 1 \le n$ gives $h \le n - 1$, and $n \le 2^{h+1} - 1$ gives $\log_2 n < h + 1$; since h is an integer, $h \ge \lfloor \log_2 n \rfloor$. □

A sum of consecutive powers of two is one less than the next power: $1 + 2 + 4 + 8 = 16 - 1$. Only perfect trees reach the upper bound (15 nodes at height 3, 31 at height 4); chains reach the lower one (4 nodes at height 3).

Example 9.4

What are the smallest and largest possible heights of a binary tree with 1,000 nodes?

1. **Largest.** A chain: $n - 1 = 999$.
2. **Smallest.** Height h allows at most $2^{h+1} - 1$ nodes: 511 for $h = 8$, too few; 1,023 for $h = 9$, enough. So 9, as the formula says: $\lfloor \log_2 1000 \rfloor = \lfloor 9.97 \rfloor = 9$.
3. **Interpretation.** A root-to-leaf walk takes about 10 steps in a balanced tree and about 1,000 in a chain.

That is why these relations matter. Searching for a key or inserting a node follows one path from the root toward a leaf, costing $O(h)$. A balanced tree has $h \approx \log_2 n$, about 20 levels for a million nodes ($2^{20} \approx 10^6$); a skewed tree has $h = n - 1$, as slow as a linked list. Balancing search trees closes this gap.

Common mistake

Writing the maximum node count for height h as 2^h or $2^h - 1$. 2^h bounds only the bottom level; summing all levels gives $2^{h+1} - 1$. $2^h - 1$ is the formula for height measured in nodes and creeps in when conventions are mixed.

9.2.5 Leaves and internal nodes in a full binary tree

A full binary tree always has exactly one more leaf than internal nodes; the 7-node perfect tree in Table 9.7 has 3 internal nodes and 4 leaves.

Theorem 9.4 (Leaves in a full binary tree)

A nonempty full binary tree with i internal nodes has $l = i + 1$ leaves. Hence $n = i + l = 2i + 1$ is always odd.

To see why, note that every full binary tree grows from a single node by one repeated operation: since a leaf cannot get a single child, it gets *two* at once. Figure 9.17 shows the growth.

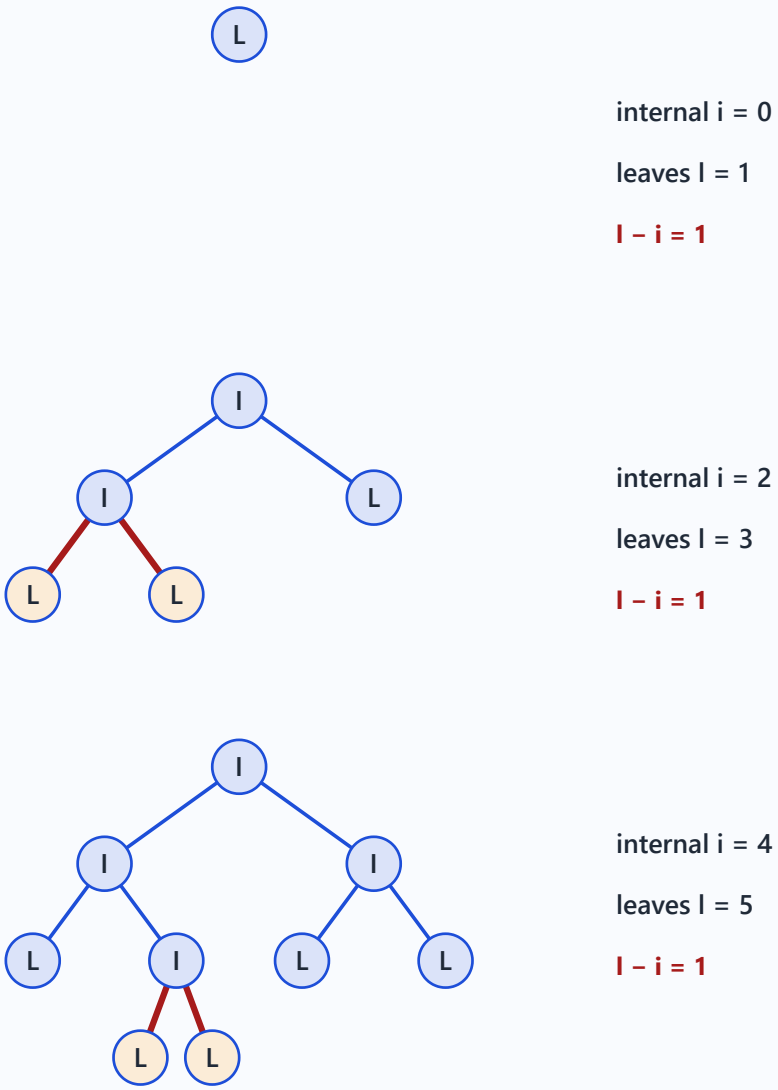


Figure 9.17: A full binary tree growing (I: internal node, L: leaf, or external node; orange: leaves just added). Top: one node, $i = 0$, $l = 1$. Middle: after two expansions, $i = 2$, $l = 3$. Bottom: after four, $i = 4$, $l = 5$. Always $l - i = 1$.

Proof. By induction on the number of nodes. *Base:* a one-node tree has $i = 0$, $l = 1$. *Step:* take the deepest leaf. Its parent has two children because the tree is full, and the sibling is a leaf too, or it would have deeper children. Deleting both leaves leaves a smaller full tree with $l' = i' + 1$. Putting them back turns one leaf into an internal node ($i = i' + 1$) and gives $l = l' - 1 + 2 = l' + 1$, so $l - i = l' - i' = 1$. □

So no full binary tree has an even number of nodes, 6 for instance, and the node count fixes the rest: $l = (n + 1)/2$ and $i = (n - 1)/2$. A full tree with 15 nodes has 8 leaves and 7 internal nodes.

Applications use this directly. In an expression tree with binary operators, operators are internal nodes and numbers leaves, so an expression always has one more number than operators: $(3 + 4) \times (5 - 2)$ has 3 operators and 4 numbers. In Huffman coding every internal node joins two subtrees, so k characters give k leaves and $k - 1$ internal nodes.

Common mistake

Applying $l = i + 1$ to every binary tree. Nodes with one child add internal nodes but no leaves: a three-node chain has 2 internal nodes and 1 leaf. In general, if i_2 nodes have two children, there are $i_2 + 1$ leaves, regardless of one-child nodes.

Counting edges proves this. With i_1 one-child and i_2 two-child nodes, there are $n - 1 = l + i_1 + i_2 - 1$ edges; since every edge leads down to a child, there are also $i_1 + 2i_2$. Equating them gives $l = i_2 + 1$; in a full tree $i_1 = 0$ and we recover the theorem.

9.3 Representations

On paper a tree is nodes and lines; in memory it is either an array of consecutive cells or objects pointing to one another. The choice of **representation** decides how fast we reach parents and children, how much memory we use, and how easily the tree grows.

9.3.1 Array representation

The level-by-level numbering gives a natural way to put a binary tree in an array: the root goes in the first cell, its children in the next two, the next level in the next four. A node's children and parent are then found from its index by simple arithmetic.

	1-based (root at `a[1]`)	0-based (root at `a[0]`)
Left child	$2i$	$2i + 1$
Right child	$2i + 1$	$2i + 2$
Parent	$\lfloor i / 2 \rfloor$	$\lfloor (i - 1) / 2 \rfloor$

Table 9.8: Index relations for the node at index i in the array representation.

The 1-based formulas are simpler, so many books leave cell 0 empty. Why they work: the first node at depth d gets index 2^d, since the levels above hold $2^d - 1$ nodes. If j nodes precede a node on its level, their $2j$ children precede its children on the next, which puts the left child of index i at $2i$ and the right child beside it. Integer division by 2 maps both $2i$ and $2i + 1$ back to i.

The 0-based formulas shift indices by one: 0-based i is 1-based $i + 1$, whose left child $2i + 2$ is 0-based $2i + 1$; the others follow likewise.

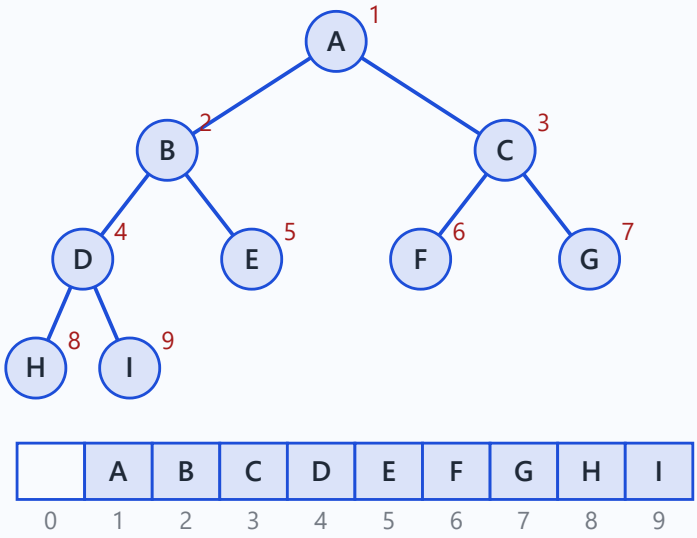


Figure 9.18: A nine-node binary tree, A to I, in a 1-based array: A at index 1, B at 2, ..., I at 9; cell 0 unused. Red numbers are indices.

In Figure 9.18, E is at index 5. Its children would be at 10 and 11, beyond the 9 nodes, so E is a leaf; its parent is at $\lfloor 5 / 2 \rfloor = 2$, B. Nodes carry no references: parent and child are implicit in the position, and each access is $O(1)$ arithmetic.

Arrays suit **complete** binary trees best: their nodes fill indices 1 to n exactly, wasting no cells, which is why binary heaps almost always live in arrays. In other trees empty positions become empty cells; we compute the cost below.

9.3.2 Code trace: index arithmetic and the path to the root

The code turns the 1-based formulas into three methods and prints every ancestor of a node up to the root.

```
int left(int i)    { return 2 * i; }
int right(int i)   { return 2 * i + 1; }
int parent(int i)  { return i / 2; }
void pathToRoot(char[] a, int i) {  // from i up to the root
  while (i >= 1) {
    System.out.print(a[i] + " ");
    i = parent(i);
  }
}
```

In `parent`, `i / 2` is Java integer division, which rounds down: exactly $\lfloor i / 2 \rfloor$. `pathToRoot(a, i)` prints `a[i]` and moves `i` to the parent while `i ≥ 1`. The root's "parent" is $\lfloor 1 / 2 \rfloor = 0$, which stops the loop; that is why an empty cell 0 is handy, meaning "no parent."

Call / step	Computation	Result
`left(2)`	$2 \cdot 2 = 4$	`a[4] = D`, B's left child
`right(2)`	$2 \cdot 2 + 1 = 5$	`a[5] = E`, B's right child
`parent(2)`	$2 / 2 = 1$	`a[1] = A`, the root
`left(4)`, `right(4)`	8 and 9	H and I, D's children
`left(5)`, `right(5)`	10 and 11 > n	outside the array: E is a leaf
`parent(9)`	$9 / 2 = 4$ (4.5 rounded down)	D; `parent(8)` is also 4
`pathToRoot` pass 1	$i = 9 \geq 1$	prints I; `i = 4`
Pass 2	$i = 4 \geq 1$	prints D; `i = 2`
Pass 3	$i = 2 \geq 1$	prints B; `i = 1`
Pass 4	$i = 1 \geq 1$	prints A; `i = 0`
Loop ends	$0 \geq 1$ is false	output: `I D B A`

Table 9.9: Index computations on the array of Figure 9.18 and a trace of `pathToRoot(a, 9)` ($n = 9$).

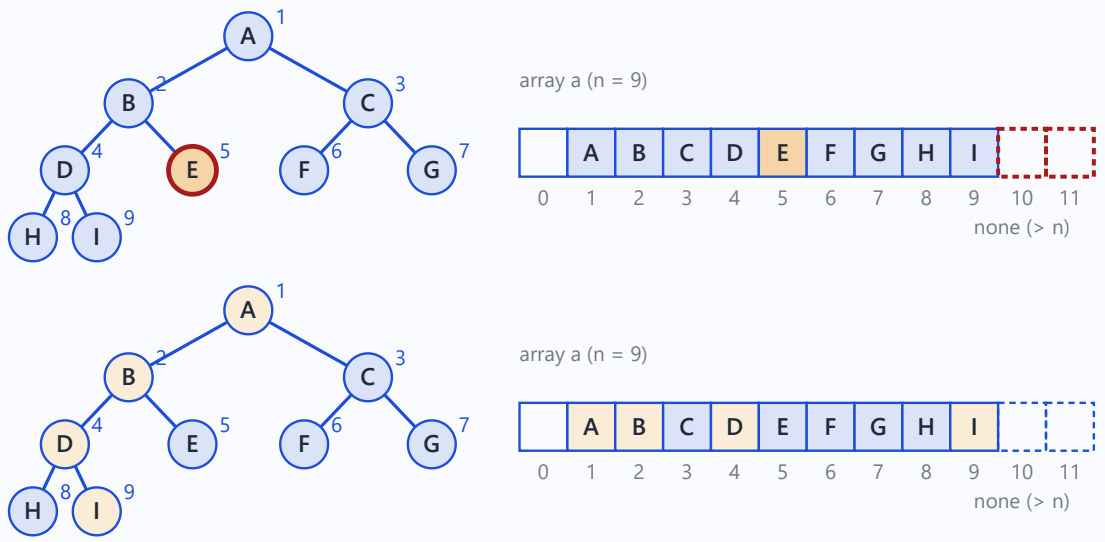


Figure 9.19: Top: E's child indices ($i = 5$) 10 and 11 fall outside the array; dashed cells mean "none." Bottom: `pathToRoot(a, 9)` has printed I, D, B, A.

Two points stand out. First, a child index must be checked against the length: the left child exists if $2i \leq n$, the right if $2i + 1 \leq n$; otherwise the code reads past the end and throws `ArrayIndexOutOfBoundsException`. In an incomplete tree the cell may also be empty. Second, siblings share a parent: 8 and 9 both lead to 4, since integer division drops the right child's extra 1.

Each pass roughly halves the index, so the loop runs depth + 1 times, at most $\lfloor \log_2 n \rfloor + 1$ in a complete tree: $O(\log n)$. This is also why heap operations are fast: moving an element up takes at most height-many steps, like this loop.

Common mistake

Mixing up the 1-based and 0-based formulas. With the root at `a[0]`, the left child is $2i + 1$; with $2i$, the root's "left child" is $2 \cdot 0 = 0$, itself. In a 0-based array, index 6 has parent $\lfloor (6 - 1) / 2 \rfloor = 2$ and left child $2 \cdot 6 + 1 = 13$, not the 3 and 12 of the 1-based formulas. Fix the layout before writing code.

9.3.3 Wasted space in the array

The array reserves a cell for every position, whatever the tree's shape: a node's index depends only on its position, so if the tree is not complete, the empty positions before it become empty cells.

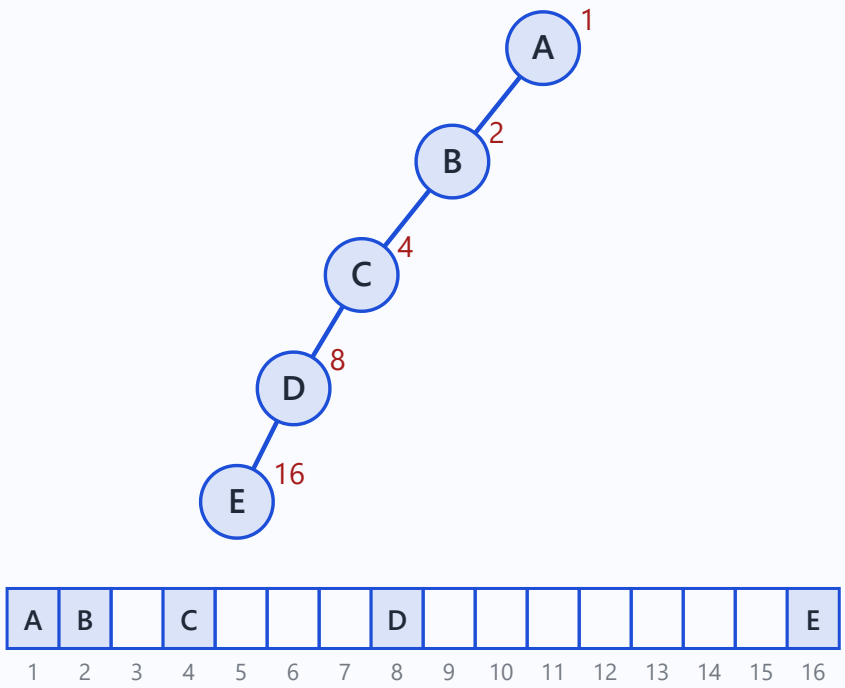


Figure 9.20: A left-skewed five-node chain in an array: A at 1, B at 2, C at 4, D at 8, E at 16. Only 5 of 16 cells are used.

In Figure 9.20 each node is the previous one's left child, so the index doubles: 1, 2, 4, 8, 16, and five nodes need 16 cells, 11 empty. A right-skewed chain is worse: with $2i + 1$ the indices are 1, 3, 7, 15, 31.

In general n nodes may need up to $2^n - 1$ cells: a chain of height $n - 1$ forces room for every position of the perfect tree of that height. A right-skewed chain of 30 nodes needs over a billion cells. Such exponential waste makes arrays impractical for trees of unpredictable shape.

Pros	Cons
No references; parent and child indices computed in *O(1)*.	Large waste for sparse or skewed trees: up to $2^n - 1$ cells.
No gaps in complete trees: the tightest use of memory.	Fixed size; a growing tree must be copied into a larger array.

Table 9.10: Pros and cons of the array representation.

9.3.4 Linked representation

Alternatively, each node is an object pointing to its children — the linked-list node adapted to trees, with two references, left and right, instead of one.

```
public class TreeNode {
    int data;
    TreeNode left;    // left child (null if none)
    TreeNode right;   // right child (null if none)

    public TreeNode(int data) {
        this.data = data;   // left = right = null
    }
}
```

`data` holds the node's value; `left` and `right` point to the children, or are `null`. The constructor sets only `data`; unassigned reference fields default to `null`, so a new node starts out as a leaf.

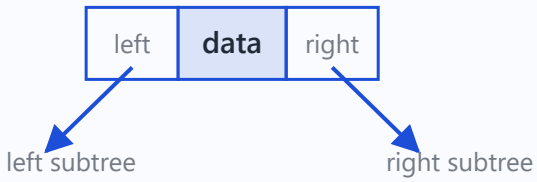


Figure 9.21: A linked tree node: left reference, data, right reference, the references pointing to the roots of the left and right subtrees.

A small tree can be built by hand:

```
// build a small tree rooted at 9
TreeNode root = new TreeNode(9);
root.left = new TreeNode(2);
root.right = new TreeNode(3);
root.left.left = new TreeNode(4);
```

This creates root 9, its children 2 and 3, and, through `root.left.left`, 4 as the left child of 2. Node 2 has no right child, and 3 has no children.

Memory goes only to existing nodes: however skewed the tree, n nodes are n objects. Adding a node means creating an object and pointing a reference at it, $O(1)$ once the spot is known. The price is access: we must follow references from the root, with no jumping by index, and moving to a parent requires an extra `parent` field.

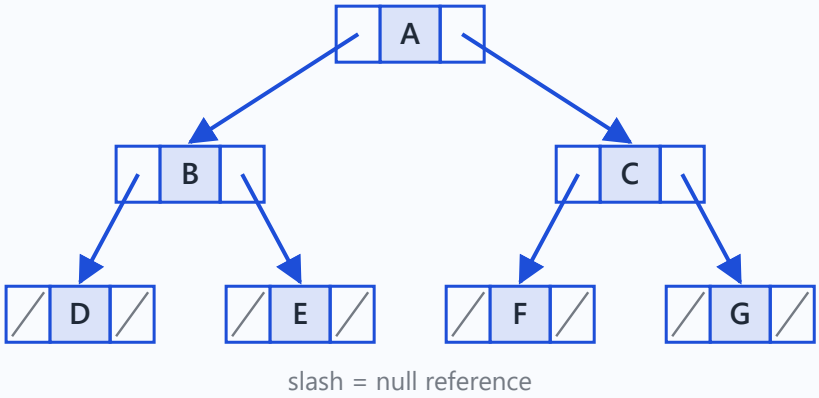


Figure 9.22: A seven-node tree in the linked representation. Each node's reference cells point to its children; slashed cells are `null`, as are both references of the leaves D, E, F, G.

Count the references in Figure 9.22: seven nodes have 14 fields. Six are non-null, one per edge (Theorem 9.1), and 8 are `null`. This is no accident.

Theorem 9.5 (Number of null references)

In the linked representation with left and right references, every binary tree with $n \geq 1$ nodes has exactly $n + 1$ `null` child references.

Proof. The $2n$ references include one non-null reference per edge, from parent to child, so $n - 1$ are non-null and $2n - (n - 1) = n + 1$ are null. □

So more than half of a linked binary tree's reference fields are always empty: 51 of 100 in a 50-node tree. **Threaded** binary trees put this space to use, pointing empty left and right references to the previous and next node in inorder.

Common mistake

Accessing a node's fields without a `null` check: `root.left.left` throws `NullPointerException` if `root.left` is `null`. Before following a reference in tree code, ask whether it can be `null`; recursive methods usually check once, in the base case on the first

line.

9.3.5 Array or links?

The two representations complement each other, as Table 9.11 shows.

Criterion	Array representation	Linked representation
Memory	Minimal for a complete tree; up to $2^n - 1$ cells wasted for a sparse one	2 extra references per node; no waste
Child / parent access	*O(1)*, by index arithmetic	Child *O(1)*; parent needs an extra reference
k*th node on level *i	*O(1)*: index $2^i + k - 1$ (1-based, $k \geq 1$)	Requires walking from the root
Growth	Copied into a larger array when full	Adding a node is *O(1)* (if the position is known)
Best suited to	Complete binary trees (heaps)	General trees of changing shape (BSTs, expression trees)

Table 9.11: Array and linked representations compared.

The third row is another advantage of arrays: level i starts at index 2^i, so its kth node is at $2^i + k - 1$; in Figure 9.18, the 3rd node of level 2 is F, at $4 + 3 - 1 = 6$. With links we must walk level by level.

The rule is short: arrays for trees that stay complete, links for unpredictable shapes. A heap stays complete through insertions and deletions, so it lives in an array; a BST takes whatever shape its keys give it, so it uses links.

9.3.6 Representing general trees

With unlimited children, a fixed number of reference fields will not do. Two solutions are common.

```
// 1) List of children
class GeneralNode {
    int data;
    List<GeneralNode> children
        = new ArrayList<>();
}
// 2) First child – next sibling
class SiblingNode {
    int data;
    SiblingNode firstChild;
    SiblingNode nextSibling;
}
```

In the first (`GeneralNode`), each node keeps its children in a list that grows as needed, and a `for-each` over `children` mirrors the ADT's `children(p)` . It is intuitive, but every node carries a list object.

In the second (`SiblingNode`), each node has two references: its **first child** (`firstChild`) and its **next sibling** (`nextSibling`). To visit a node's children, go down to the first and walk right along the siblings; the child list becomes a linked list spread over the nodes.

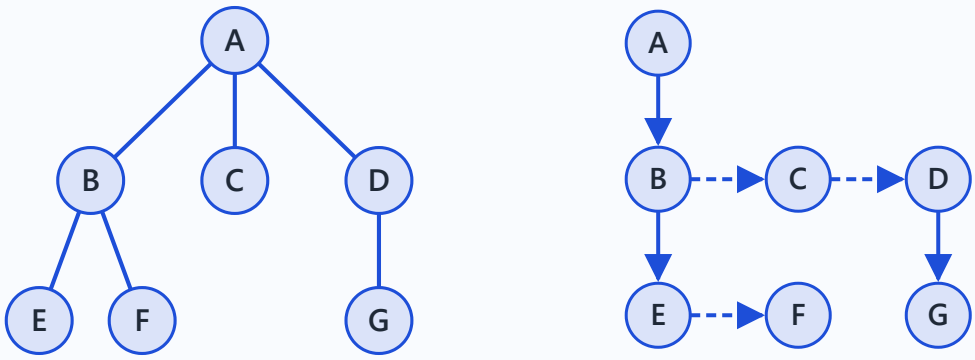


Figure 9.23: Left: a general tree; A's children are B, C, D, B's are E and F, D's is G. Right: the same tree in first-child (solid arrow) – next-sibling (dashed arrow) form.

Example 9.5

Give both references of every node in the first-child – next-sibling form of Figure 9.23.

1. **A.** The root: `firstChild = B` , `nextSibling = null` .
2. **B, C, D.** A's children form the chain B → C → D. B: first child E, next sibling C. C: no children, next sibling D. D: first child G, no next sibling.
3. **E, F, G.** E: no children, next sibling F. F and G: both `null` .
4. **Count.** Of 14 fields, six are non-null — A→B, B→E, B→C, C→D, D→G, E→F — which is $n - 1 = 6$.

Call the first-child reference “left” and the next-sibling reference “right,” and any general tree becomes a binary tree. Rotated 45 degrees, the right half of Figure 9.23 is one: B is A's left child; E and C are B's children; D is C's right child, F is E's right child, G is D's left child. Binary tree algorithms thus extend to general trees, and a forest becomes one binary tree if its roots are treated as siblings.

Use the child list if a node's kth child is accessed often: `children.get(k)` is $O(1)$, against k steps along the sibling chain. The sibling form uses a small fixed amount of memory per node and needs no list object.

9.4 Tree traversals

An array or list has one natural order of visiting: front to back. A tree has none. At a node, do we process the node or its subtrees first? Which subtree first? Or do we go level by level? Each answer gives a traversal, and each traversal naturally solves a different problem.

9.4.1 What is a traversal?

Definition 9.5

A **traversal visits** every node of a tree exactly once, in a particular order. A "visit" is whatever work is done at the node: printing it, adding it to a sum, copying or deleting it.

Traversals come in two families. **Depth-first** traversals follow a branch to its end before backing up to the next; the three members differ only in *when* the root is visited relative to its subtrees. The **breadth-first** traversal, **level order**, scans level by level: the root, its children, its grandchildren.

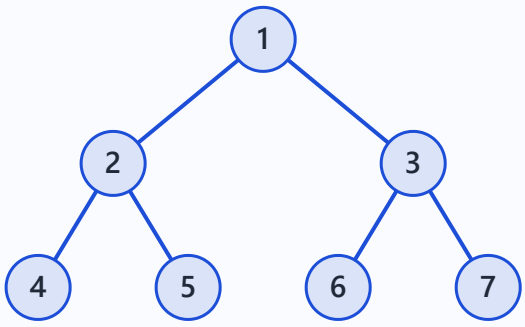


Figure 9.24: The example tree for traversals: the perfect binary tree numbered 1 to 7, with root 1, children 2 and 3, and leaves 4 to 7.

Traversal	Order	Output	Typical use
Preorder	root → left → right	`1 2 4 5 3 6 7`	Copying a tree, prefix expressions
Inorder	left → root → right	`4 2 5 1 6 3 7`	Visiting a BST in sorted order
Postorder	left → right → root	`4 5 2 6 7 3 1`	Deletion, computing size and height
Level order	level by level, left to right	`1 2 3 4 5 6 7`	Level-based operations

Table 9.12: The four traversals and their outputs on the tree of Figure 9.24.

The depth-first traversals are named for where the root falls: before (pre), between (in), or after (post) its subtrees, the left subtree always preceding the right. They follow the recursive definition — act at the root, apply the same traversal to each subtree — so when working one out by hand, treat

each subtree as a small tree of its own.

9.4.2 Preorder traversal

A **preorder** traversal has three steps:

1. Visit the root.
2. Traverse the left subtree in preorder.
3. Traverse the right subtree in preorder.

```
visit(root);
preorder(root.left);
preorder(root.right);
```

A node precedes all its descendants in the output. On Figure 9.24: visit 1; enter the left subtree and visit its root 2; visit 2's subtrees, 4 and 5. Then the right subtree: 3, 6, 7. Output: `1 2 4 5 3 6 7`.

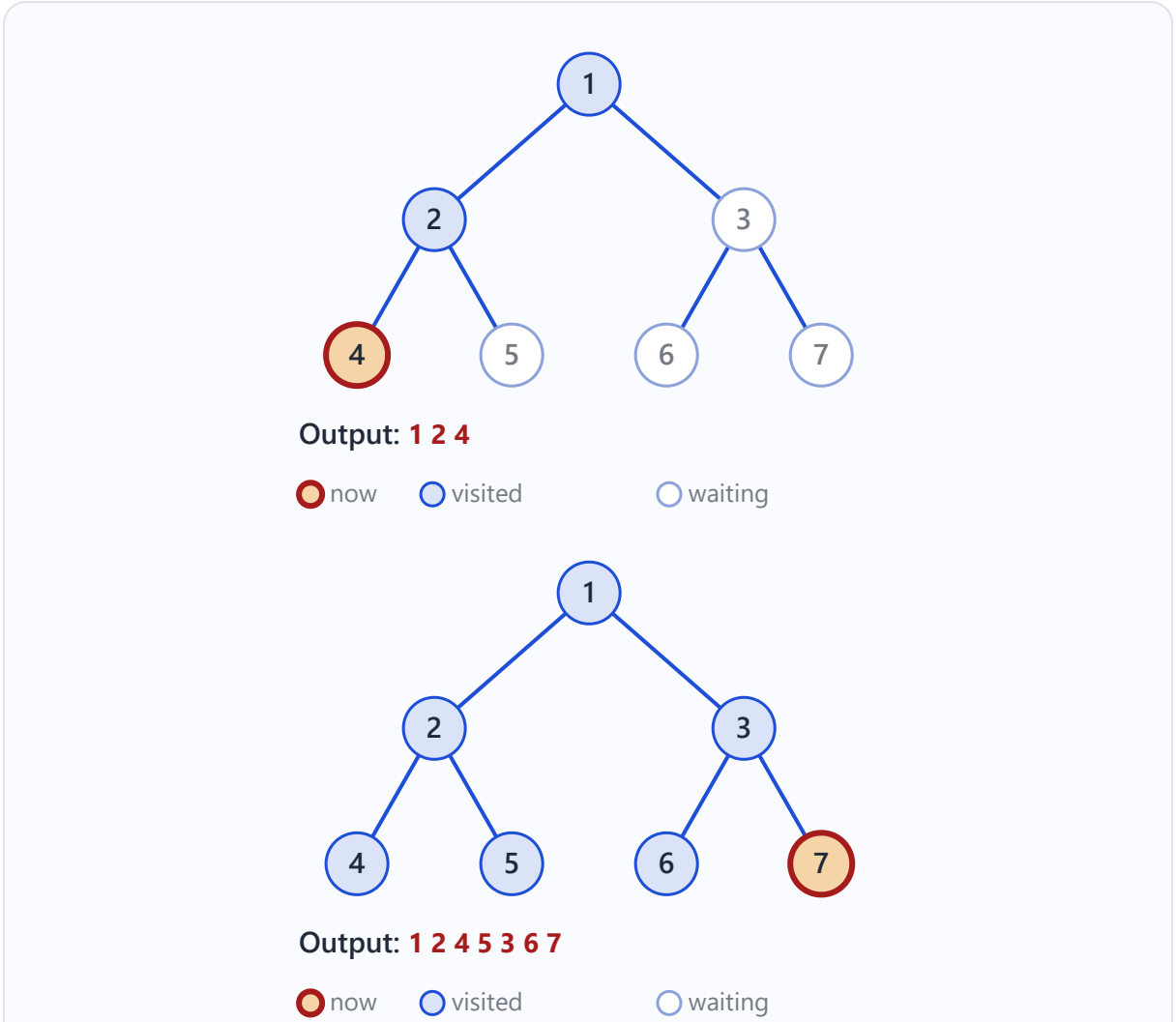


Figure 9.25: Preorder. Left: after 1 and 2, the leaf 4 is visited; output `1 2 4`. Right: done; `1 2 4 5 3 6 7`.

Preorder fits whenever a node must be handled before its children. **Copying a tree** is the clearest case: children cannot be attached before their parent's copy exists. **Writing a tree to a file** uses preorder too, so the reader meets each node after its parent and can rebuild the tree. A table of contents is a preorder traversal, and on expression trees preorder gives **prefix** notation, operators before operands.

9.4.3 Inorder traversal

In an **inorder** traversal the root is visited between its subtrees:

1. Traverse the left subtree in inorder.
2. Visit the root.
3. Traverse the right subtree in inorder.

```
inorder(root.left);
visit(root);
inorder(root.right);
```

A node comes after its whole left subtree and before its right one, so the output starts not with the root but with the leftmost node. On Figure 9.24 we go from 1 to 2 to 4; 4's subtrees are empty, so 4 is visited, then 2, then 5. With the left subtree done, 1 is visited; in the right subtree we again go far left: 6, 3, 7. Output: `4 2 5 1 6 3 7`.

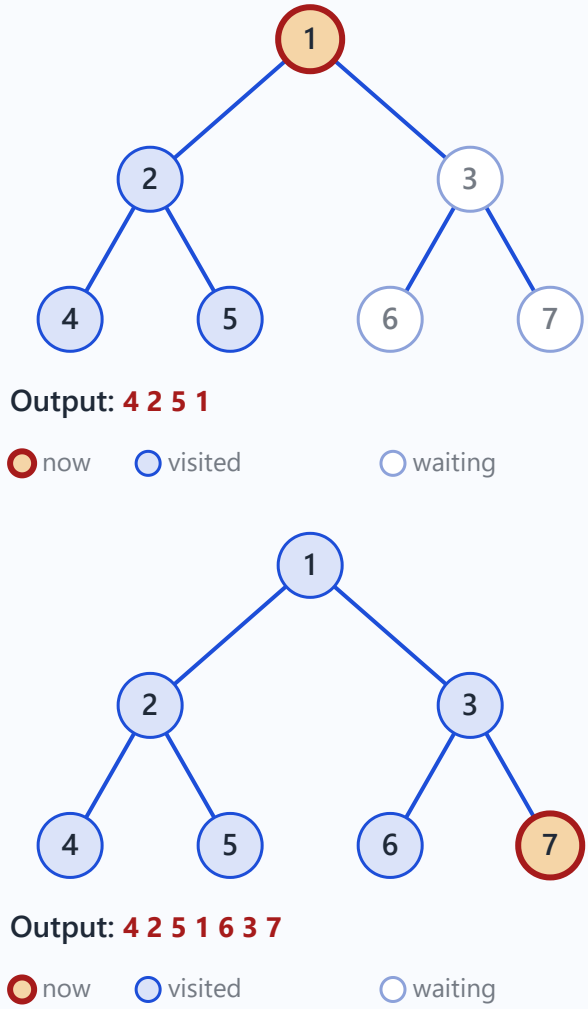


Figure 9.26: Inorder. Left: the left subtree (4, 2, 5) is done and the root 1 visited. Right: done; `4 2 5 1 6 3 7`.

Geometrically, inorder lists the nodes in their left-to-right order in the drawing, as if each were dropped straight down onto a line: 4, 2, 5, 1, 6, 3, 7 in Figure 9.24.

Its key use is in **binary search trees**. Smaller keys lie left of every node and larger ones right, so inorder outputs the keys **in increasing order**; conversely, a binary tree with sorted inorder output (and distinct keys) is a BST. On expression trees inorder gives **infix** notation, minus the parentheses.

Common mistake

Starting inorder at the root. The root follows its entire left subtree; the first node written is the leftmost, reached by going left as far as possible. Also remember that the leftmost node may have a right subtree: after writing it, enter that subtree and again go far left.

9.4.4 Postorder traversal

In a **postorder** traversal the root is visited after both subtrees are finished:

1. Traverse the left subtree in postorder.
2. Traverse the right subtree in postorder.
3. Visit the root.

```
postorder(root.left);
postorder(root.right);
visit(root);
```

Every node follows all its descendants, and the root comes last. On Figure 9.24: the leftmost leaf 4, then 5, then 2; the right subtree gives 6, 7, 3; the root 1 is last. Output: `4 5 2 6 7 3 1`.

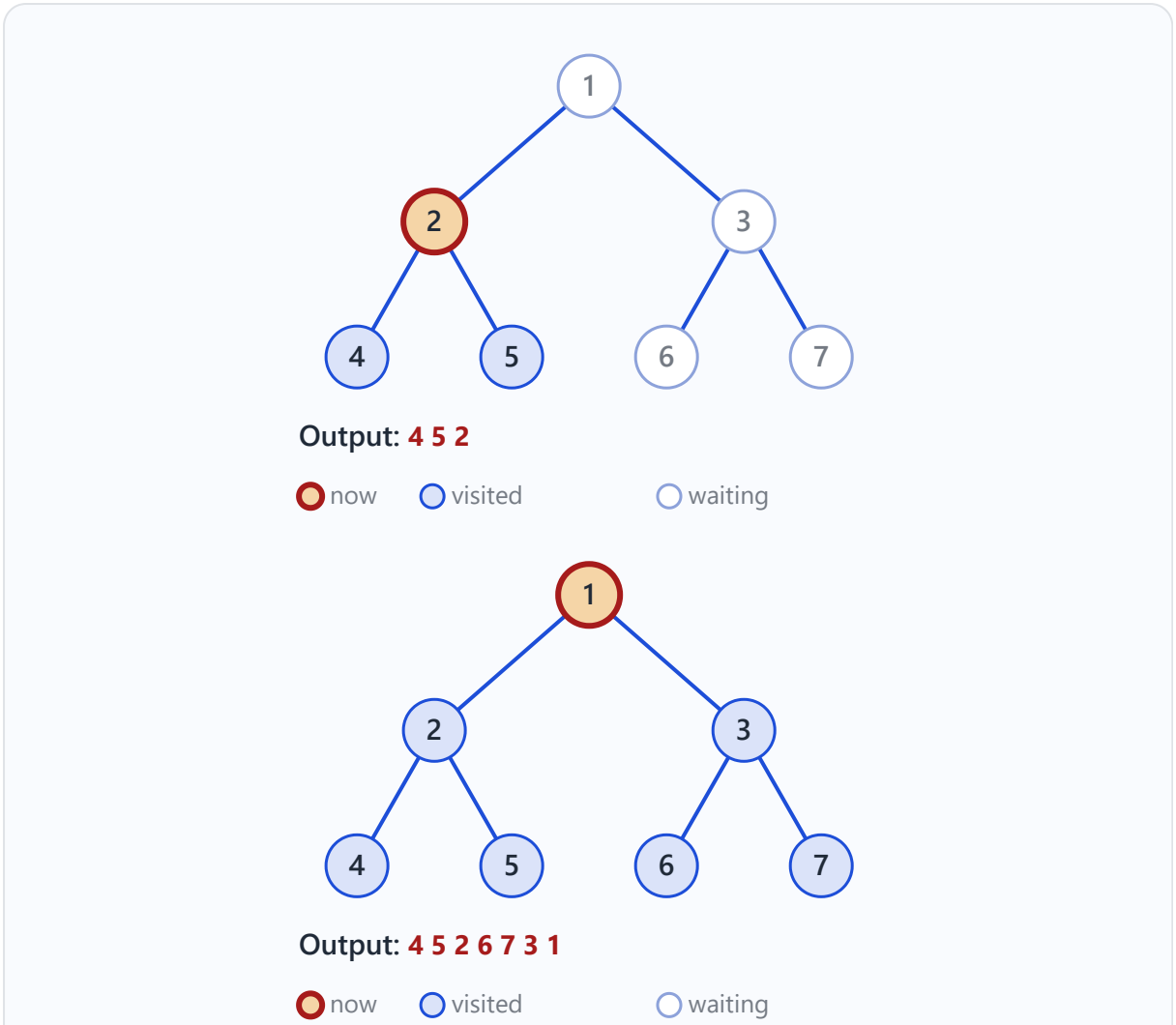


Figure 9.27: Postorder. Left: 2 visited after 4 and 5; output `4 5 2`. Right: the root last; `4 5 2 6 7 3 1`.

Postorder fits whenever a node's work depends on its children's results. **Deleting a tree** must remove children first, or the references to them are lost; likewise a nonempty folder cannot be deleted before its contents. A folder's total disk usage, a tree's **node count**, and its **height** are all computed in postorder. On expression trees postorder gives **postfix** notation, the order of evaluation.

A quick check relates the outputs: the root is *first* in preorder, *last* in postorder, and between its subtrees in inorder — in Table 9.12, 1 starts, ends, and splits the outputs. This observation also underlies rebuilding a tree from two traversals.

9.4.5 Three traversals, one tour: the Euler tour

One picture unites the three depth-first traversals. Walk around the tree from the left of the root, keeping the edges at your left hand. This **Euler tour** passes every edge twice, down and up, and meets every node three times: on its left, below it, and on its right.

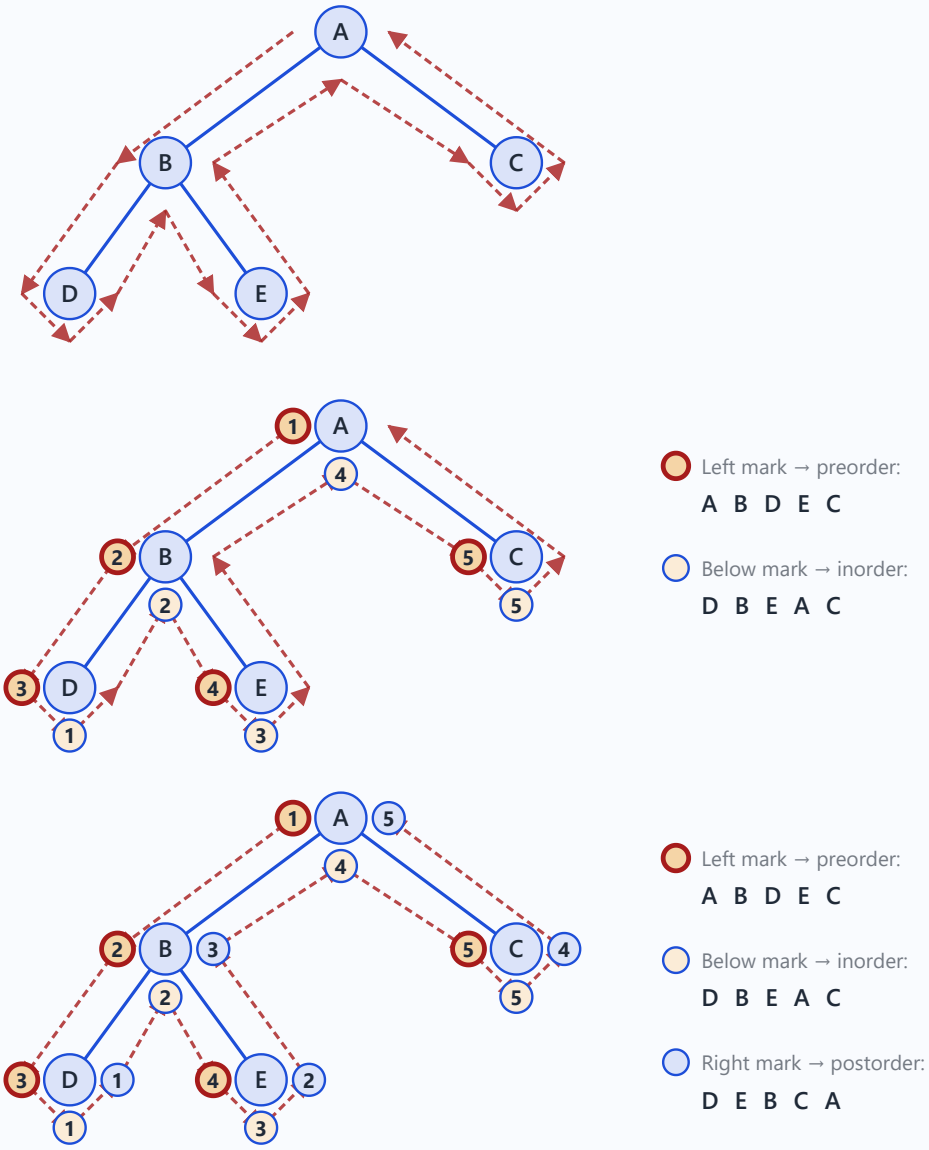


Figure 9.28: The Euler tour of a five-node tree (root A; B's children D and E; C is A's right child). Top: the path. Middle: passes on the left (preorder: A B D E C) and below (inorder: D B E A C) numbered. Bottom: passes on the right (postorder: D E B C A) added.

The pass at which we write a node picks the traversal.

- On its **left**, on first arrival: **preorder**, `A B D E C`.
- **Below** it, between its subtrees: **inorder**, `D B E A C`.
- On its **right**, after both subtrees: **postorder**, `D E B C A`.

At a leaf the three passes are consecutive, since its subtrees are empty, so leaves keep the same relative order in all three traversals (D, E, C); only internal nodes move. The three traversals are *moments of one tour*. The tour takes *2(n − 1)* edge steps, *O(n)* — another reason all three are *O(n)*.

Some jobs need all three passes. Writing an expression fully parenthesized opens "(" on entering a subtree, writes the operator below, and closes ")" on leaving.

9.4.6 Level-order traversal

A **level-order** traversal scans each level left to right, top down — the tree version of **breadth-first search** (BFS) on graphs.

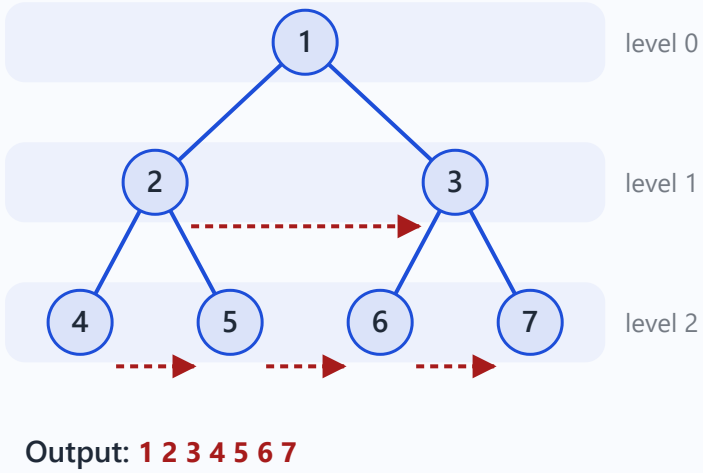


Figure 9.29: Level order: 1 on level 0; 2, 3 on level 1; 4–7 on level 2. Dashed arrows show the scan; output `1 2 3 4 5 6 7`.

Level order ignores subtree structure: after 2 it moves to 2's sibling 3, not its child 4, and 2's children wait until level 1 is done. Recursion, which finishes one subtree before starting the next, cannot express this jumping back and forth.

The solution is a first-in, first-out **queue**. A visited node's children join the back of the queue, so earlier nodes' children come out earlier, and each level enters the queue before the next.

Level order suits jobs where levels mean something: printing level by level, per-level sums or maxima, the tree's width, or the leaf nearest the root. For that last job it is ideal: the first leaf to leave the queue is among the nearest, and the traversal can stop without visiting deeper nodes.

Common mistake

Confusing level order with preorder. Both start at the root, yet on Figure 9.24 they differ by the third element: `1 2 4` versus `1 2 3`. Preorder dives into a branch; level order finishes siblings first. Writing level order with a stack is the same confusion in code: the output becomes depth-first.

9.5 Coding the traversals

Traversals are defined recursively, and their shortest code is recursive. Behind the recursion sits an invisible structure, the **call stack**: each call pushes a **stack frame** holding its parameters, local variables, and return point. We first trace the recursive traversals on the call stack, then replace it with an explicit `Deque` used as a stack, and finally code level order with a queue.

All code uses `TreeNode` from the linked representation. The recursive traces use the four-node tree rooted at 9: 9's children are 2 and 3, and 2's left child is 4 (height 2).

9.5.1 Recursive preorder and the call stack

```
public void preorder(TreeNode root) {
  if (root == null) {
    return;
  }
  System.out.print(root.data + " ");
  preorder(root.left);
  preorder(root.right);
}
```

If `root == null` we are in an empty subtree and return at once: the **base case**. Otherwise the method prints the root (line 5), recurses on the left subtree (line 6), then on the right (line 7), and at the closing brace (line 8) its frame is popped.

A `null` base case keeps the code simple — leaves need no special case, since their two calls return at once — at the price of one call per `null` child. By Theorem 9.5 there are *n + 1* of those, so *2n + 1* calls in all: 9 for our tree.

Step	Call stack	Event	Output
1	preorder(9)	prints 9	9
2	9 · ln 6 | preorder(2)	prints 2	9 2
3	9 · ln 6 | 2 · ln 6 | preorder(4)	prints 4	9 2 4
4	… | 4 · ln 6 | preorder(null)	base case, return	9 2 4
5	… | 4 · ln 7 | preorder(null)	base case, return	9 2 4
6	9 · ln 6 | 2 · ln 6 | preorder(4)	4 finishes (line 8)	9 2 4
7	9 · ln 6 | 2 · ln 7 | preorder(null)	2's right is empty	9 2 4
8	9 · ln 6 | preorder(2)	2 finishes	9 2 4
9	9 · ln 7 | preorder(3)	prints 3	9 2 4 3
10–11	9 · ln 7 | 3 · ln 6 / ln 7 | preorder(null)	both of 3's children are empty	9 2 4 3
12	9 · ln 7 | preorder(3)	3 finishes	9 2 4 3

Step	Call stack	Event	Output
13	preorder(9)	9 finishes, the stack empties	9 2 4 3

Table 9.13: Trace of `preorder(9)`. The call stack is written bottom to top; "ln 6" and "ln 7" give the line from which a call was made.

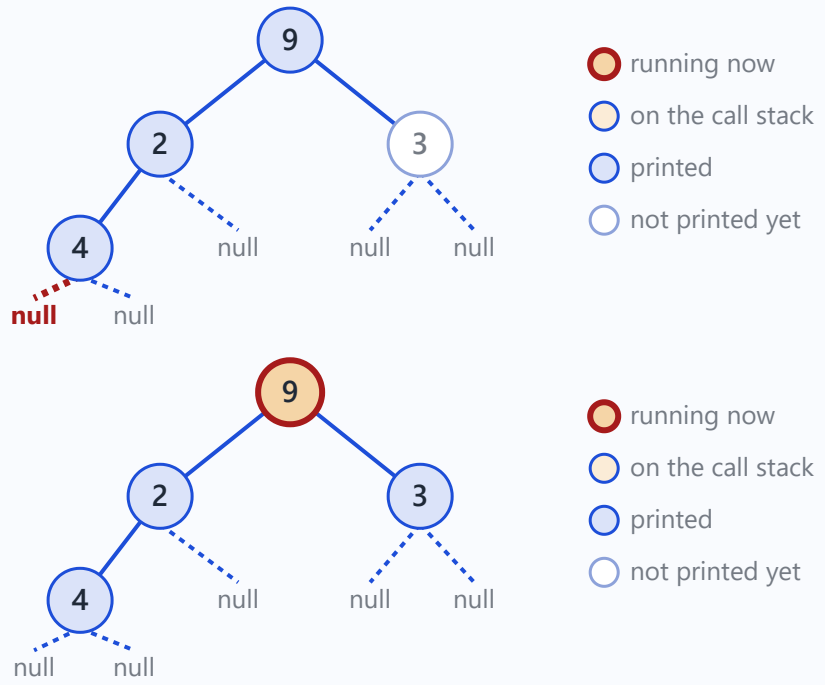


Figure 9.30: Recursive preorder. Top, step 4: `preorder(4.left)` is at the base case; frames for 9, 2, 4 wait. Bottom, last step: output `9 2 4 3`.

Note the stack depth. At steps 4 and 5 it peaks at four frames — 9, 2, 4, and 4's `null` child. Depth is set by the path to the deepest `null` child: *h + 1* nodes plus one `null` call, at most *h + 2* frames; here *2 + 2 = 4*.

Time is *O(n)*, one constant-work call per node and per `null`. Extra memory is the stack depth, *O(h)*: *O(log n)* if balanced, *O(n)* if skewed.

9.5.2 Recursive inorder

```
public void inorder(TreeNode root) {
  if (root == null) {
    return;
  }
  inorder(root.left);
  System.out.print(root.data + " ");
  inorder(root.right);
}
```

Only the print has moved, between the two calls. A call no longer prints on entry; it traverses its whole left subtree first and prints on line 6, after the left call returns.

Step	Call stack	Event	Output
1–3	9 · ln 5 | 2 · ln 5 | inorder(4)	9, 2, 4 are called; none printed yet	—
4	… | 4 · ln 5 | inorder(null)	4's left is empty	—
5	9 · ln 5 | 2 · ln 5 | inorder(4)	left done: prints 4	4
6–7	… | 4 · ln 7 | inorder(null)	4's right is empty; 4 finishes	4
8	9 · ln 5 | inorder(2)	left done: prints 2	4 2
9–10	9 · ln 5 | 2 · ln 7 | inorder(null)	2's right is empty; 2 finishes	4 2
11	inorder(9)	left done: prints 9	4 2 9
12–13	9 · ln 7 | 3 · ln 5 | inorder(null)	3 is called; 3's left is empty	4 2 9
14	9 · ln 7 | inorder(3)	prints 3	4 2 9 3
15–17	inorder(9)	3's right is empty; 3 and 9 finish	4 2 9 3

Table 9.14: Trace of `inorder(9)`. Left calls are made from line 5, right calls from line 7.

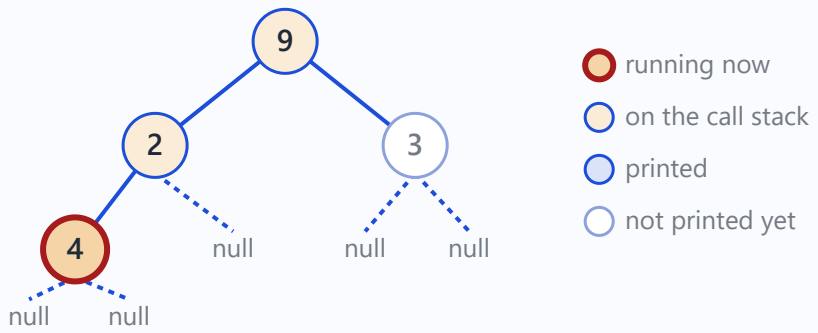


Figure 9.31: Recursive inorder, step 5: 4's empty left subtree is done and 4 printed; frames for 2 and 9 wait for their left calls.

The output is `4 2 9 3`: 9 after its left subtree, before its right. 9's frame, created at step 1, waits on line 5 until step 11 to print. A frame on the stack marks unfinished work.

9.5.3 Recursive postorder

```
public void postorder(TreeNode root) {
  if (root == null) {
    return;
  }
  postorder(root.left);
  postorder(root.right);
  System.out.print(root.data + " ");
}
```

Now the print follows both calls (lines 5 and 6), on line 7: a value is printed only after both subtrees are fully traversed.

Step	Call stack	Event	Output
1–3	9 · ln 5 | 2 · ln 5 | postorder(4)	9, 2, 4 are called	—
4	… | 4 · ln 5 | postorder(null)	4's left is empty	—
5	… | 4 · ln 6 | postorder(null)	4's right is empty	—
6–7	9 · ln 5 | 2 · ln 5 | postorder(4)	both subtrees done: prints 4; 4 finishes	4
8	9 · ln 5 | 2 · ln 6 | postorder(null)	2's right is empty	4
9–10	9 · ln 5 | postorder(2)	prints 2; 2 finishes	4 2
11–13	9 · ln 6 | 3 · ln 5 / ln 6 | postorder(null)	3 is called; both children empty	4 2
14–15	9 · ln 6 | postorder(3)	prints 3; 3 finishes	4 2 3
16–17	postorder(9)	prints 9; the stack empties	4 2 3 9

Table 9.15: Trace of `postorder(9)`. Left calls are made from line 5, right calls from line 6.

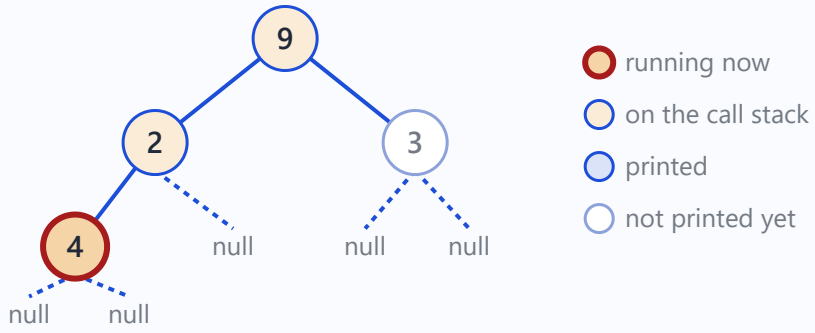


Figure 9.32: Recursive postorder, step 6: both (empty) subtrees of 4 are done and 4 printed; 9 and 2 are not yet.

The output is `4 2 3 9`. Side by side, the three traces make and return the same calls in the same order — 9, 2, 4, `null`, `null`, … — and differ only in where the print falls. This is the Euler tour in code: entering a call is passing on the node's left, the gap between calls passing below, and leaving

passing on its right.

> **Common mistake**
>
> Forgetting the base case. Without `if (root == null) return;`, the first leaf calls `preorder(root.left)` with `null` and reading `root.data` throws `NullPointerException`; putting the check *after* the print fails the same way. Handle the empty tree before touching a node's fields.

9.5.4 Preorder with a stack

The system's call stack is limited: with Java's defaults a `StackOverflowError` comes after a few thousand to a few tens of thousands of frames, so recursion crashes on a chain of hundreds of thousands of nodes. An explicit `java.util.Deque` stack does the same job in heap memory, filling up only when memory runs out.

```
public void preorderIterative(TreeNode root) {
  if (root == null) return;
  Deque<TreeNode> stack = new ArrayDeque<>();
  stack.push(root);
  while (!stack.isEmpty()) {
    TreeNode temp = stack.pop();
    System.out.print(temp.data + " ");
    if (temp.right != null)
      stack.push(temp.right);  // right first
    if (temp.left != null)
      stack.push(temp.left);   // left on top
  }
}
```

Preorder is the easiest: a node precedes its children, so it can be printed as soon as it is popped. Each pass pops, prints, and pushes the children, **right first, then left**, so the left child sits on top of the LIFO stack and its whole subtree is done before the right child's turn.

Pass	pop	Pushed	Stack after the pass	Output
start	—	1	[1]	—
1	1	3, 2	[3, 2]	1
2	2	5, 4	[3, 5, 4]	1 2
3	4	—	[3, 5]	1 2 4
4	5	—	[3]	1 2 4 5
5	3	7, 6	[7, 6]	1 2 4 5 3
6	6	—	[7]	1 2 4 5 3 6

Pass	pop	Pushed	Stack after the pass	Output
7	7	—	[]	1 2 4 5 3 6 7

Table 9.16: Trace of `preorderIterative` on the tree of Figure 9.24. The stack is written bottom to top.

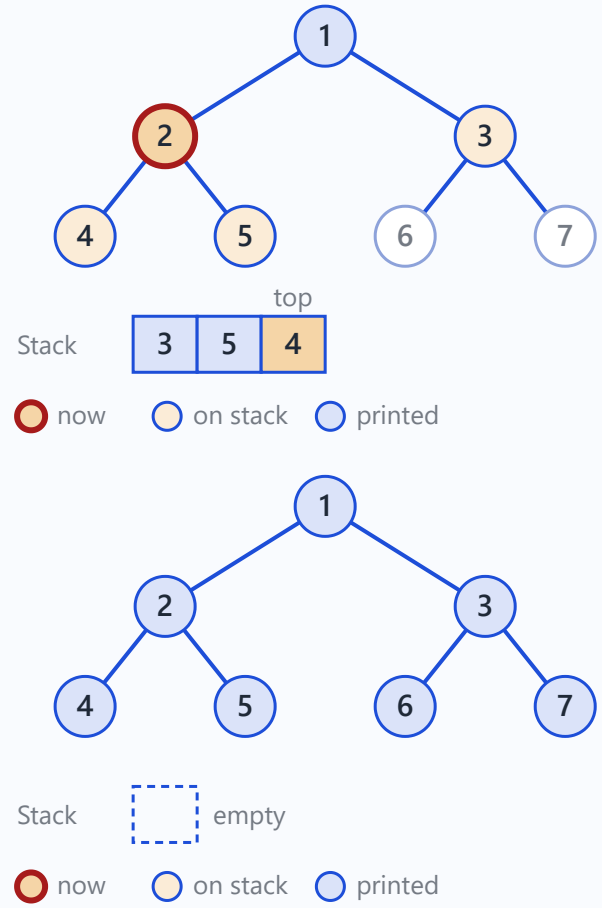


Figure 9.33: Preorder with a stack. Left: after pass 2, with 1 and 2 printed, the stack holds 3, 5, 4. Right: the stack is empty; the output `1 2 4 5 3 6 7` matches the recursive version.

The stack is the recursion's to-do list: after pass 2, [3, 5, 4] means "4's subtree, then 5's, then 3's," information the recursive version kept in the lines where frames 2 and 9 waited.

Each node is pushed and popped once: n passes, $O(n)$. The stack holds at most one waiting right sibling per level plus the top node, here at most 3 (after pass 2): $O(h)$ extra memory.

Common mistake

Pushing left before right. The right child then pops first and the output becomes root, right, left: `1 3 7 6 2 5 4` on Figure 9.24. Push right first so left stays on top.

9.5.5 Inorder with a stack

In inorder a node cannot be printed when pushed, since its left subtree comes first; the stack holds nodes "to print later." Two moves alternate: while `temp` is a node, push it and go left; when `temp` is `null`, the top node's left subtree is done, so pop it, print it, and go to its right subtree.

```
public void inorderIterative(TreeNode root) {
  if (root == null) return;
  Deque<TreeNode> stack = new ArrayDeque<>();
  TreeNode temp = root;
  while (!stack.isEmpty() || temp != null) {
    if (temp != null) {
      stack.push(temp);
      temp = temp.left;
    } else {
      temp = stack.pop();
      System.out.print(temp.data + " ");
      temp = temp.right;
    }
  }
}
```

The loop condition has two parts, `!stack.isEmpty() || temp != null`, because right after the root is printed the stack may be empty while `temp` points to the untraversed right subtree. The trace, on the tree where 1's children are 2 and 3, 2's are 4 and 5, and 3 has only a left child 6, includes this moment.

Step	Move	Stack	`temp`	Output
0	start	[]	1	—
1	push 1, go left	[1]	2	—
2	push 2, go left	[1, 2]	4	—
3	push 4, go left	[1, 2, 4]	null	—
4	pop 4, print, go right	[1, 2]	null	4
5	pop 2, print, go right	[1]	5	4 2
6	push 5, go left	[1, 5]	null	4 2
7	pop 5, print, go right	[1]	null	4 2 5
8	pop 1, print, go right	[]	3	4 2 5 1
9	push 3, go left	[3]	6	4 2 5 1
10	push 6, go left	[3, 6]	null	4 2 5 1
11	pop 6, print, go right	[3]	null	4 2 5 1 6
12	pop 3, print, go right	[]	null	4 2 5 1 6 3

Table 9.17: Trace of `inorderIterative` (tree: 1(2(4, 5), 3(6, —))). Stack bottom to top.

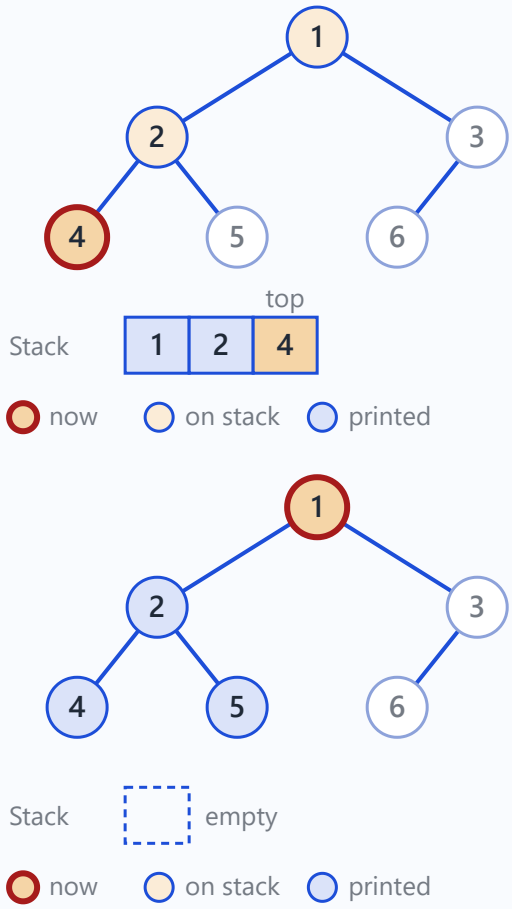


Figure 9.34: Inorder with a stack. Left, step 3: 1, 2, 4 on the stack, `temp = null`, nowhere left to go. Right, step 8: 1 printed, stack empty, but `temp` is 3, so the loop continues.

At the end of step 8 the stack is empty; with only `!stack.isEmpty()` the loop would stop and 3 and 6 would never print. `temp != null` catches the waiting right subtree. The loop ends only when both are exhausted, after step 12, with output `4 2 5 1 6 3`.

Each node is pushed and popped once and each `null` child reached once: *O(n)*. The stack holds the nodes on the path to `temp` with unfinished left subtrees, at most *h + 1*: *O(h)* extra memory.

9.5.6 Postorder with a stack

Postorder is hardest: a node prints only when *both* subtrees are done. A node on top of the stack has finished its left subtree, but we cannot tell about the right, so the code looks with `peek` before popping.

```
public void postorderIterative(TreeNode root) {
  TreeNode current = root;
  Deque<TreeNode> stack = new ArrayDeque<>();
  while (current != null || !stack.isEmpty()) {
    if (current != null) {
      stack.push(current);
      current = current.left;
    } else {
      TreeNode temp = stack.peek().right;
      if (temp == null) {
        temp = stack.pop();
        System.out.print(temp.data + " ");
        while (!stack.isEmpty()
           && temp == stack.peek().right) {
          temp = stack.pop();
          System.out.print(temp.data + " ");
        }
      } else {
        current = temp;
      }
    }
  }
}
```

The logic has three parts.

- **Going left.** If `current` is not `null`, push it and go left, as in inorder.
- **Entering the right subtree.** Otherwise look at the top node's right child (`stack.peek().right`). If it exists, the node must wait: set `current` to it and enter that subtree.
- **Printing and climbing.** With no right child, the top node is finished: pop and print it. The inner `while` then looks up: if the printed node is the new top's right child, that parent is finished too and is popped and printed, and so on up the chain.

`temp == stack.peek().right` tests reference equality — the same object? — asking "are we returning from this parent's right subtree?" If so, the parent prints; if the printed node was a left child, the parent's right subtree is still ahead and the loop stops. The trace uses the tree where 1's children are 2 and 3, 2's are 4 and 5, and 3 has only a right child, 6.

Step	Event	Stack	Output
1–3	push 1, 2, 4; `current = 4.left = null`	[1, 2, 4]	—
4	`peek() = 4`, no right child: pop and print 4	[1, 2]	4
5	inner loop: 4 is not 2's right child (2.right = 5); stop	[1, 2]	4
6	`peek() = 2`, right child 5: `current = 5`	[1, 2]	4
7	push 5; `current = null`	[1, 2, 5]	4
8	`peek() = 5`, no right child: print 5	[1, 2]	4 5

Step	Event	Stack	Output
9	inner loop: 5 is 2's right child: print 2	[1]	4 5 2
10	inner loop: 2 is not 1's right child (1.right = 3); stop	[1]	4 5 2
11	`peek() = 1`, right child 3: `current = 3`	[1]	4 5 2
12	push 3; `current = 3.left = null`	[1, 3]	4 5 2
13	`peek() = 3`, right child 6: `current = 6`	[1, 3]	4 5 2
14	push 6; `current = null`	[1, 3, 6]	4 5 2
15	`peek() = 6`, no right child: print 6	[1, 3]	4 5 2 6
16	inner loop: 6 is 3's right child: print 3	[1]	4 5 2 6 3
17	inner loop: 3 is 1's right child: print 1	[]	4 5 2 6 3 1
18–19	stack empty: inner loop stops; `current` is `null`, outer loop ends	[]	4 5 2 6 3 1

Table 9.18: Trace of `postorderIterative` (tree: 1(2(4, 5), 3(—, 6))). Stack bottom to top.

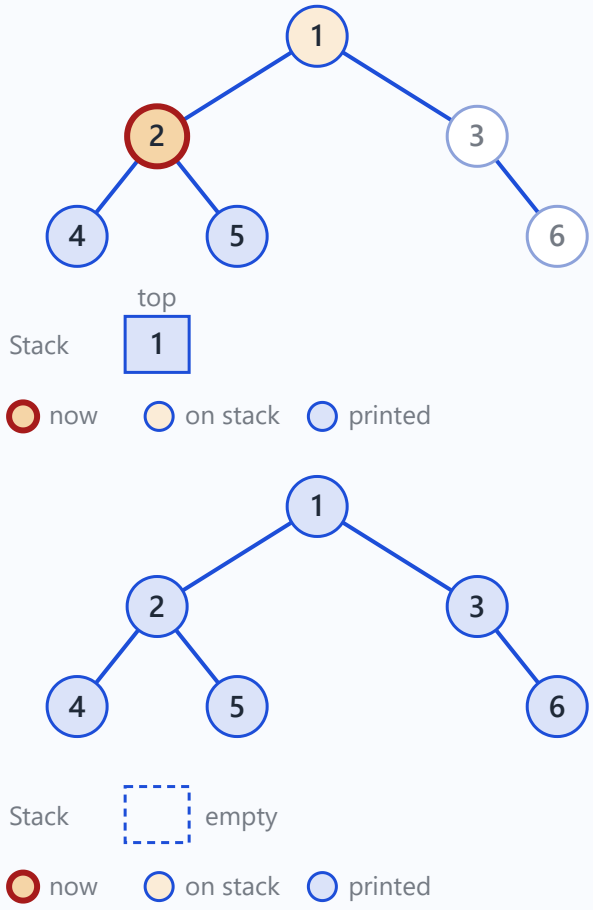


Figure 9.35: Postorder with a stack. Left, step 9: after 5, the inner loop saw 5 is 2's right child and printed 2; only 1 remains. Right: done; `4 5 2 6 3 1`.

The end of the trace shows the inner loop at work: printing 6 at step 15 completes 3 (6 is its right child) and then 1 (3 is its right child) — one leaf finishing a whole chain of parents. 3's missing left child is harmless: at step 12 `current = 3.left` is simply `null`.

The cost is *O(n)*: each node is pushed and popped once, and every `peek` leads to a pop or a right subtree. The stack holds the current root-to-node path, at most *h + 1* nodes. A shortcut exists: stack-based preorder with the pushes reversed (left, then right) outputs root, right, left, and reversing that with a second stack gives left, right, root — easy, but *O(n)* extra memory.

9.5.7 Level order with a queue

```
public void levelOrder(TreeNode root) {
  if (root == null) return;
  Queue<TreeNode> queue = new LinkedList<>();
  queue.add(root);
  while (!queue.isEmpty()) {
    TreeNode temp = queue.poll();
    System.out.print(temp.data + " ");
    if (temp.left != null)
      queue.add(temp.left);
    if (temp.right != null)
      queue.add(temp.right);
  }
}
```

This mirrors stack-based preorder with three changes: a `Queue` (a `LinkedList`) replaces the stack, `add` (to the back) and `poll` (from the front) replace `push` and `pop`, and children are added in natural order, left first, since FIFO returns them in that order.

The trace uses the tree rooted at 4, with children 3 and 7; 3's children are 5 and 8, and 7 is a leaf.

Pass	poll	Added	Queue after the pass	Output
start	—	4	[4]	—
1	4	3, 7	[3, 7]	4
2	3	5, 8	[7, 5, 8]	4 3
3	7	—	[5, 8]	4 3 7
4	5	—	[8]	4 3 7 5
5	8	—	[]	4 3 7 5 8

Table 9.19: Trace of `levelOrder` (tree: 4(3(5, 8), 7)). The queue is written front to back.

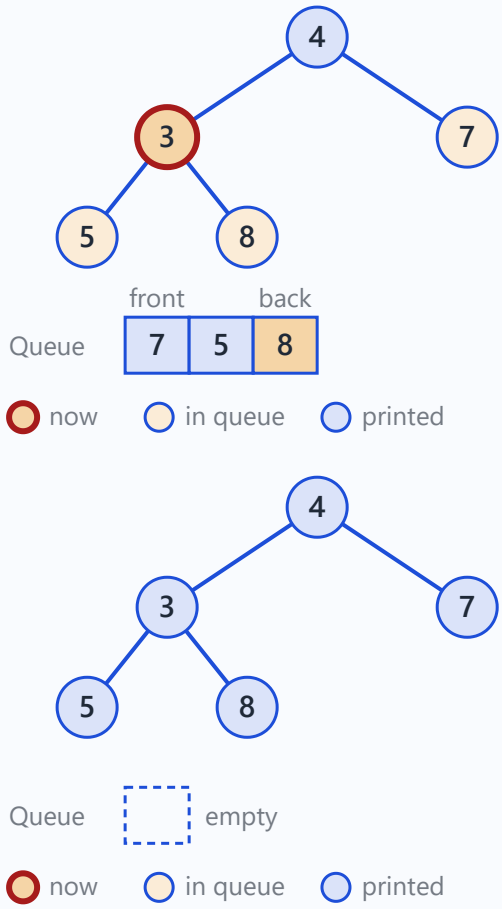


Figure 9.36: Level order with a queue. Left: after pass 2, 3's children 5 and 8 are queued behind 7. Right: the queue is empty; `4 3 7 5 8`.

Pass 2 is the essence of level order: 3's children go behind 7, which shares 3's level and is processed first. Preorder would give `4 3 5 8 7`; level order gives `4 3 7 5 8`.

Each node enters and leaves once: $O(n)$. Memory, though, depends on the **width** w, the size of the most crowded level, since the queue spans at most two consecutive levels: $O(w)$, here at most 3. A perfect tree's last level has about $n/2$ nodes, so level order needs $O(n)$ there, while stack-based inorder needs $O(\log n)$.

Common mistake

Confusing `add()` with `push()`. `LinkedList` is both a queue and a stack: `pop()` removes from the front and `push()` inserts there. Writing `push` for `add` puts children at the front, turning the queue into a stack and the output into something other than level order. Declaring the variable as `Queue<TreeNode>` rules out stack methods at compile time.

9.6 Applications of traversals

Traversals do more than print. Nearly every tree algorithm visits nodes in some order and does work at each visit; this section applies that pattern to real problems.

9.6.1 Expression trees

An arithmetic expression is naturally a tree. In *(3 + 4) × (5 − 2)* the multiplication happens last, and each of its operands is a subexpression.

Definition 9.6

An **expression tree** is a binary tree whose internal nodes are operators and whose leaves are operands (numbers or variables). With binary operators every internal node has two children, so the tree is full. An internal node's value is its operator applied to the values of its left and right subtrees.

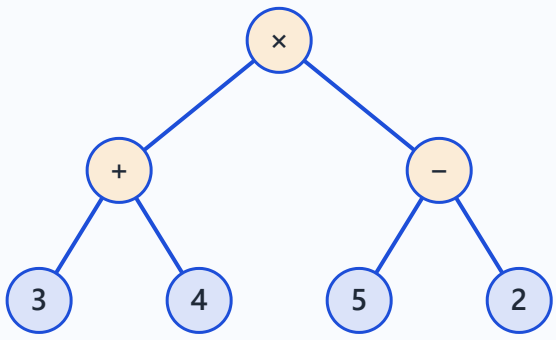


Figure 9.37: The tree of *(3 + 4) × (5 − 2)*: multiplication at the root, 3 plus 4 on the left, 5 minus 2 on the right.

The tree needs no parentheses: addition and subtraction sit deeper, so they are computed first. Precedence and grouping are built into the shape, and the three depth-first traversals give three notations.

Traversal	Output	Notation
Preorder	`× + 3 4 − 5 2`	prefix
Inorder	`3 + 4 × 5 − 2` ; with parentheses `(3 + 4) × (5 − 2)`	infix
Postorder	`3 4 + 5 2 − ×` → *7 × 3 = 21*	postfix

Table 9.20: Traversals of the tree in Figure 9.37.

Prefix and postfix need no parentheses; the order of symbols alone fixes the expression. Unparenthesized inorder does not: `3 + 4 × 5 − 2` happens to equal *3 + 20 − 2 = 21* under the usual precedence, but it is not the tree's expression. Correct infix puts every internal node's subtree in parentheses — an Euler tour writing "(" on entry and ")" on exit.

> **Common mistake**
>
> Taking the inorder output as the infix expression. Without parentheses it may describe a different tree: the tree for *((6 + 2) × 3) − (8 / 4)* gives `6 + 2 × 3 − 8 / 4`, which evaluates to *6 + 6 − 2 = 10*, while the tree's value is 22.

9.6.2 Code trace: evaluate()

Evaluation is postorder: an operator needs both operand values first. The code assumes an `ExprNode` class holding a number (`value`) at a leaf and an operator character (`op`) at an internal node.

```
int evaluate(ExprNode node) {
  if (node.left == null && node.right == null)
    return node.value;           // leaf: operand
  int a = evaluate(node.left);
  int b = evaluate(node.right);
  switch (node.op) {             // internal node: operator
    case '+': return a + b;
    case '-': return a - b;
    case '*': return a * b;
    default:  return a / b;
  }
}
```

A node with no children is an operand, and its number is returned: the base case. Otherwise the left subtree's value goes into `a` (line 4), the right's into `b` (line 5), and the `switch` applies the operator. Order matters for subtraction and division: in `a - b`, `a` is the left operand. An expression tree is full, so the one-child case never arises.

Step	Call stack	Event
1	evaluate(×)	internal node; left operand first: `evaluate(+)`
2	× · ln 4 | evaluate(+)	internal node; left operand first: `evaluate(3)`
3	× · ln 4 | + · ln 4 | evaluate(3)	leaf: returns 3
4–5	× · ln 4 | + · ln 5 | evaluate(4)	`a = 3`; right operand is a leaf: returns 4
6	× · ln 4 | evaluate(+)	`b = 4`; returns *3 + 4 = 7*
7–8	× · ln 5 | evaluate(−)	`a = 7`; right operand `evaluate(-)`
9–11	× · ln 5 | − · ln 4 / ln 5 | evaluate(5), evaluate(2)	leaves 5 and 2 return

Step	Call stack	Event
12	× · ln 5 | evaluate(−)	returns *5 − 2 = 3*
13	evaluate(×)	`b = 3`; returns *7 × 3 = 21*

Table 9.21: Trace of `evaluate(×)`. Call stack bottom to top; “ln 4” is the call for the left operand, “ln 5” for the right.

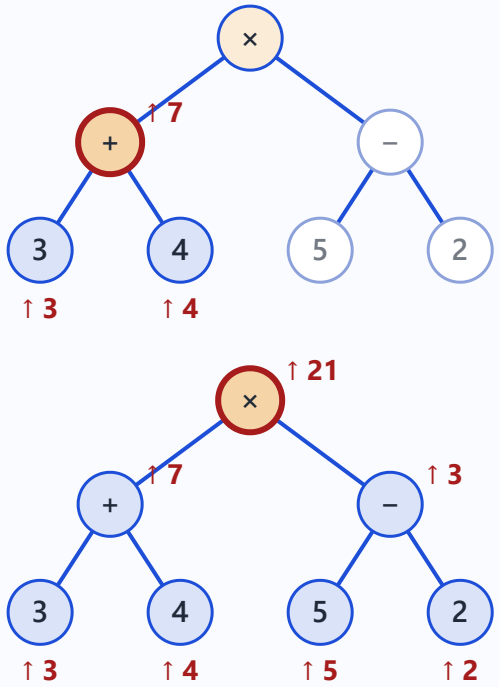


Figure 9.38: Tracing `evaluate`. Left, step 6: 3 and 4 passed up, + returned 7. Right, last step: − returned 3, the root 21. Arrowed numbers are return values.

Values flow upward, and the operators apply in the order `+`, `−`, `×` — postorder, matching the postfix form `3 4 + 5 2 − ×`. The stack-based postfix evaluator does the same work, with the call stack as its stack. Each node is processed once: *O(n)* time, *O(h)* extra memory.

9.6.3 Finding the largest value

An unordered binary tree hides its maximum anywhere, so every node must be checked. Recursively, the largest value is the largest of the root's value and the maxima of the two subtrees.

```
public int findMax(TreeNode root) {
  if (root == null) {
    return Integer.MIN_VALUE;
  }
  int result = root.data;
  int left = findMax(root.left);
  int right = findMax(root.right);
  if (left > result)
    result = left;
  if (right > result)
    result = right;
  return result;
}
```

An empty tree has no largest value, but its return value must not distort the comparison. `Integer.MIN_VALUE` (-2^{31}) never wins `left > result`, so empty subtrees have no effect. The body is postorder: subtree results first, then the root.

Call	`result` at first	`left`	`right`	Returns
`findMax(5)`	5	MIN	MIN	5
`findMax(8)`	8	MIN	MIN	8
`findMax(3)`	3	5 (> 3: `result = 5`)	8 (> 5: `result = 8`)	8
`findMax(7)`	7	MIN	MIN	7
`findMax(4)`	4	8 (> 4: `result = 8`)	7 (≤ 8)	8

Table 9.22: Trace of `findMax` (tree: 4(3(5, 8), 7)). MIN stands for `Integer.MIN_VALUE`.

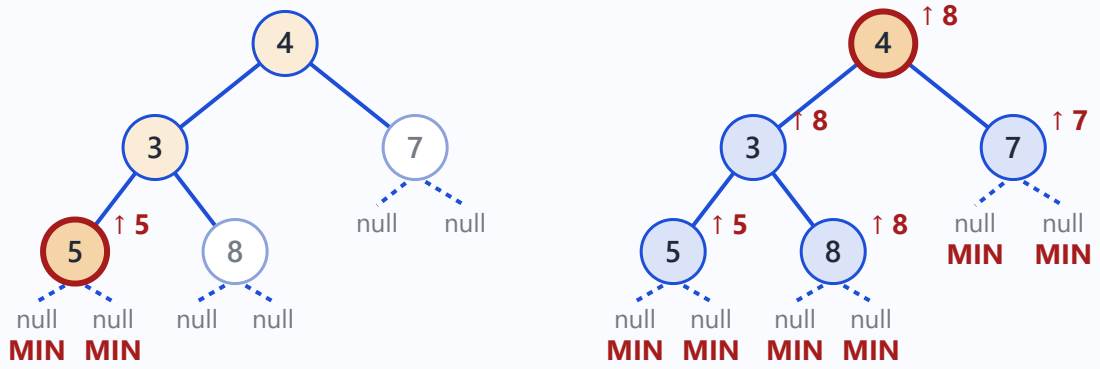


Figure 9.39: Tracing `findMax`. Left: 5's empty subtrees returned MIN, and 5 returned itself. Right: all values passed up; 3's subtree gives 8, 7 itself, the root 8.

The table lists calls in the order they *finish* — 5, 8, 3, 7, 4, postorder. With one call per node and per `null` child, 11 calls in all, time is $O(n)$. In a BST we would just go right from the root, $O(h)$; an unordered tree forces a look at every node.

Common mistake

Returning 0 for the empty tree. In an all-negative tree — nodes −5 and −3, say — the method returns 0, a value in no node. The base value must be the *identity element* of the combining operation: `Integer.MIN_VALUE` for a maximum, 0 for a sum, 1 for a product.

9.6.4 Recursive tree functions

`evaluate` and `findMax` share a pattern: a base value for the empty tree, otherwise the subtrees' results combined with the root. The pattern computes many basic properties in a few lines.

```
int size(TreeNode node) {            // number of nodes
    if (node == null) return 0;
    return 1 + size(node.left) + size(node.right);
}

int height(TreeNode node) {          // number of edges
    if (node == null) return -1;     // empty tree: -1
    return 1 + Math.max(height(node.left),
                        height(node.right));
}

int countLeaves(TreeNode node) {
    if (node == null) return 0;
    if (node.left == null && node.right == null) return 1;
    return countLeaves(node.left) + countLeaves(node.right);
}
```

- `size(node)` counts nodes: 0 for an empty tree, otherwise 1 for the root plus `size(node.left)` and `size(node.right)`.
- `height(node)` measures edges, and its base case returns **−1**. Odd as it looks, this makes a leaf's height *1 + max(−1, −1) = 0*; returning 0 would give leaves height 1, measuring nodes.
- `countLeaves(node)` has two base cases: 0 for an empty tree, 1 for a node with two `null` children. Any other node returns the sum of its subtrees' counts, not counting itself.

On the four-node tree rooted at 9 (children 2 and 3; 4 is 2's left child) they give `size = 4`, `height = 2`, and `countLeaves = 2` (leaves 4 and 3). Each processes every node once: *O(n)* time, *O(h)* stack space.

9.6.5 Code trace: size()

The trace shows values carried up from the subtrees. All recursive calls come from line 3; Java evaluates left to right, so `size(node.left)` runs before `size(node.right)`.

Step	Call	Event	Returns
1–3	`size(9)`, `size(2)`, `size(4)`	each calls its left subtree first	—
4	`size(4.left)`	empty tree	0
5–6	`size(4.right)`	left gave 0; right is empty too	0
7	`size(4)`	*1 + 0 + 0*	1
8–9	`size(2.right)`	left gave 1; right is empty	0
10	`size(2)`	*1 + 1 + 0*	2
11–15	`size(3)`	both children empty; *1 + 0 + 0*	1
16–17	`size(9)`	*1 + 2 + 1*	4

Table 9.23: Trace of `size(9)` (tree: 9(2(4, —), 3)).

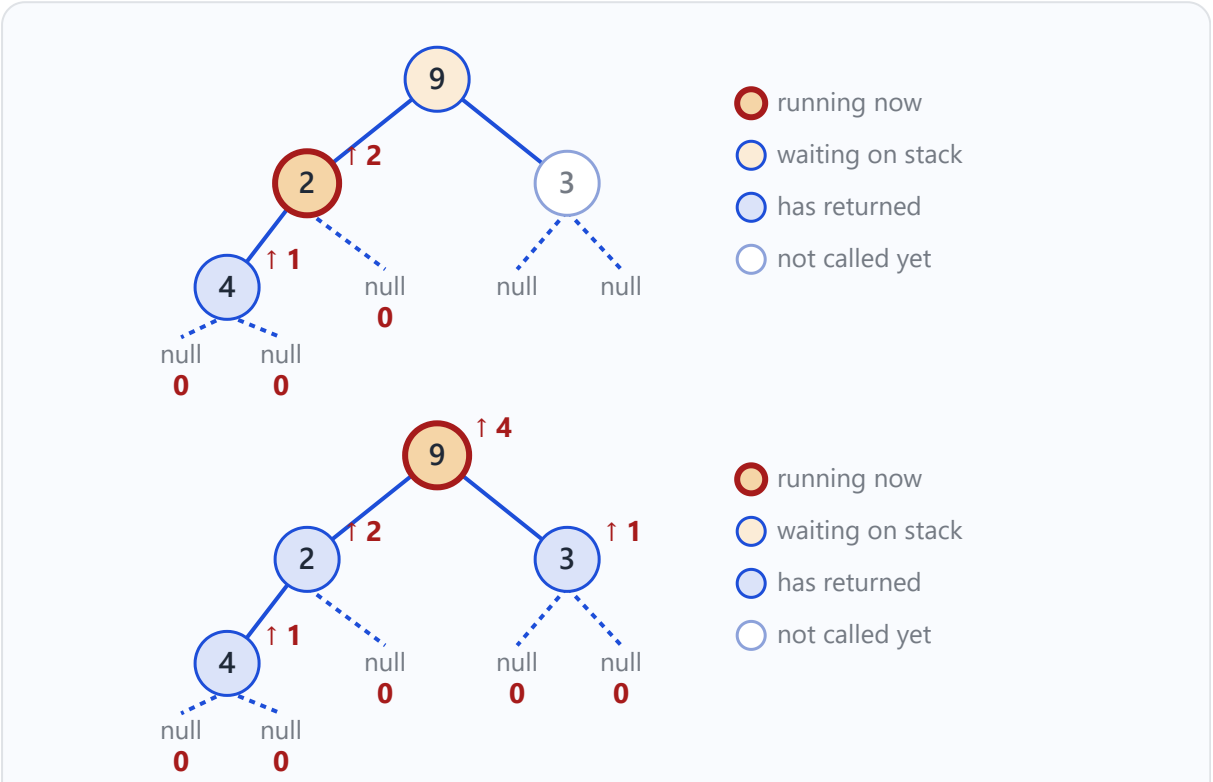


Figure 9.40: Tracing `size`. Top, step 10: 4's subtree returned 1, 2's returned 2. Bottom, last step: 3 returned 1 and the root *1 + 2 + 1 = 4*. The 0s under `null` children are the empty trees' returns.

Nine calls are made, for four nodes and five `null` children: $2n + 1 = 9$. Each node returns its subtree's size — 1 for 4, 2 for 2, 1 for 3, 4 for 9 — so one postorder pass sizes every subtree. Stored in the nodes, these sizes answer questions like "what is the *k*th smallest key?" quickly.

9.6.6 Code trace: height()

Here the left call comes from line 3 and the right from line 4, the two arguments of `Math.max` being on separate lines. Same tree.

Step	Call	Computation	Returns
1–3	`height(9)`, `(2)`, `(4)`	each calls its left subtree first	—
4–6	`height(4.left)`, `height(4.right)`	empty trees	−1, −1
7	`height(4)`	*1 + max(−1, −1)*	0
8–9	`height(2.right)`	empty tree	−1
10	`height(2)`	*1 + max(0, −1)*	1
11–16	`height(3)`	*1 + max(−1, −1)*	0
17	`height(9)`	*1 + max(1, 0)*	2

Table 9.24: Trace of `height(9)` (tree: 9(2(4, —), 3)).

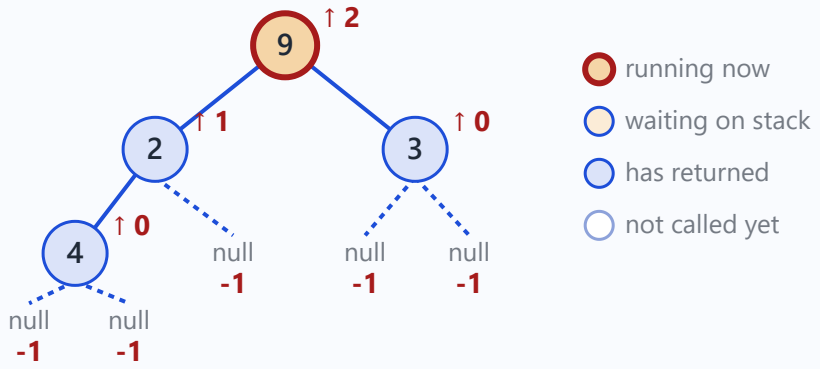


Figure 9.41: The last step of the `height` trace. Empty trees returned −1, the leaves 4 and 3 returned 0; 2 has height 1 and the root *1 + max(1, 0) = 2*.

The result is 2, the two edges of 9–2–4. The call structure equals that of `size`; only the combination changes, maximum instead of sum. That is the pattern's power: changing the base value and the combining operation yields node sums, minima, occurrence counts, and much more from one skeleton.

9.6.7 Code trace: countLeaves()

`countLeaves` returns at a leaf without further calls, so the leaves' `null` children are never visited and the trace is shorter.

Step	Call	Event	Returns
1	`countLeaves(9)`	not a leaf; left first	—
2	`countLeaves(2)`	not a leaf; left first	—
3	`countLeaves(4)`	both children `null`: a leaf	1
4	`countLeaves(2.right)`	empty tree	0
5	`countLeaves(2)`	*1 + 0*	1
6	`countLeaves(3)`	a leaf	1

Step	Call	Event	Returns
7	`countLeaves(9)`	*1 + 1*	2

Table 9.25: Trace of `countLeaves(9)` (tree: 9(2(4, —), 3)).

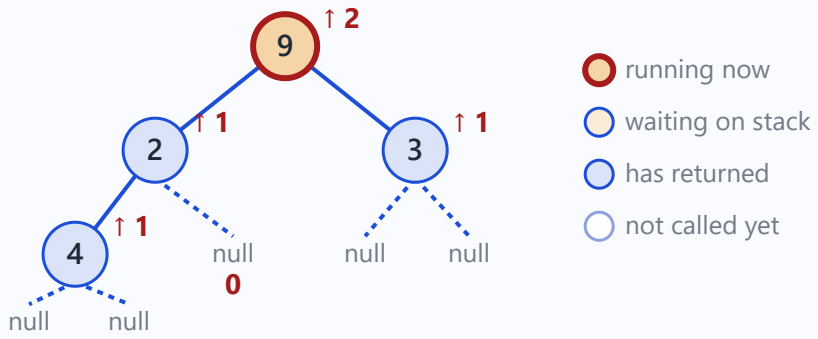


Figure 9.42: The last step of the `countLeaves` trace. Leaves 4 and 3 returned 1, 2's empty right subtree 0; the root gets *1 + 1 = 2*.

Only five calls are made — four nodes plus 2's empty right child — against nine for `size`, because the leaf check stops descent at leaves. That check is also essential: without it the method returns 0 at empty trees and never produces a 1, so every tree would have 0 leaves. A leaf is what we count, so it needs its own base case.

Common mistake

Writing only the `node == null` base case in `countLeaves`, or putting the leaf check before it. The first always returns 0; the second may read `node.left` with `node` null and throw `NullPointerException`. Keep the two base cases in the order shown.

9.6.8 Rebuilding a tree from its traversals

One traversal does not determine a tree: preorder `1 2` fits 2 as either child of 1. Two suitable traversals together, with distinct values, determine it uniquely.

Theorem 9.6 (Rebuilding from traversals)

A binary tree with distinct node values can be built in exactly one way from its inorder and preorder outputs (or from its inorder and postorder outputs). Preorder and postorder outputs, however, do not determine a tree that contains nodes with one child.

The method follows from the definitions. The first preorder element is the root; in the inorder output, the elements to its left form the left subtree and those to its right the right subtree. The left subtree's size then tells which preorder elements belong to it, and the step recurses on both subtrees. With postorder, the root is the *last* element instead.

Preorder plus postorder fixes the root but cannot separate the subtrees: `A B` and `B A` fit B as A's left or right child. Only inorder distinguishes them, writing a left child before the root and a right child after. In full trees there are no one-child nodes, so the ambiguity vanishes.

```
int p = 0; // next root in pre[]
TreeNode build(int[] pre, int[] in, int lo, int hi) {
    if (lo > hi) return null;          // empty range
    TreeNode node = new TreeNode(pre[p++]);
    int m = lo; while (in[m] != node.data) m++;
    node.left = build(pre, in, lo, m - 1);  // left part
    node.right = build(pre, in, m + 1, hi); // right part
    return node;
}
```

`build(pre, in, lo, hi)` builds the subtree for `in[lo..hi]` and returns its root, or `null` for an empty range (`lo > hi`). Otherwise `pre[p]` is the subtree's root: a node is created and `p` incremented. As a field, `p` is shared by all calls and sweeps the preorder array once. The root's index `m` in the inorder range splits it into `in[lo..m−1]` (left) and `in[m+1..hi]` (right). The left subtree must be built first, since its elements precede the right's in preorder.

Example 9.6

Build the tree from `pre = [1, 2, 4, 5, 3, 6]` and `in = [4, 2, 5, 1, 3, 6]`, starting with `build(pre, in, 0, 5)`.

1. `pre[0] = 1`. Root 1, at *m = 3* in `in`: left subtree `in[0..2] = 4, 2, 5`, right `in[4..5] = 3, 6`.
2. `pre[1] = 2`. Left subtree's root 2; *m = 1*; left range `[0..0]` (4), right `[2..2]` (5).
3. `pre[2] = 4`. 2's left child 4; *m = 0*; both ranges empty, both calls return `null`: a leaf.
4. `pre[3] = 5`. 2's right child 5; *m = 2*; a leaf. 2's subtree is done: `1.left = 2`.
5. `pre[4] = 3`. Right subtree's root 3; *m = 4*; left range `[4..3]` empty, so no left child; right range `[5..5]`.
6. `pre[5] = 6`. 3's right child 6; *m = 5*; a leaf. `1.right = 3`.
7. **Result.** 1(2(4, 5), 3(—, 6)). Check: preorder `1 2 4 5 3 6`, inorder `4 2 5 1 3 6`.

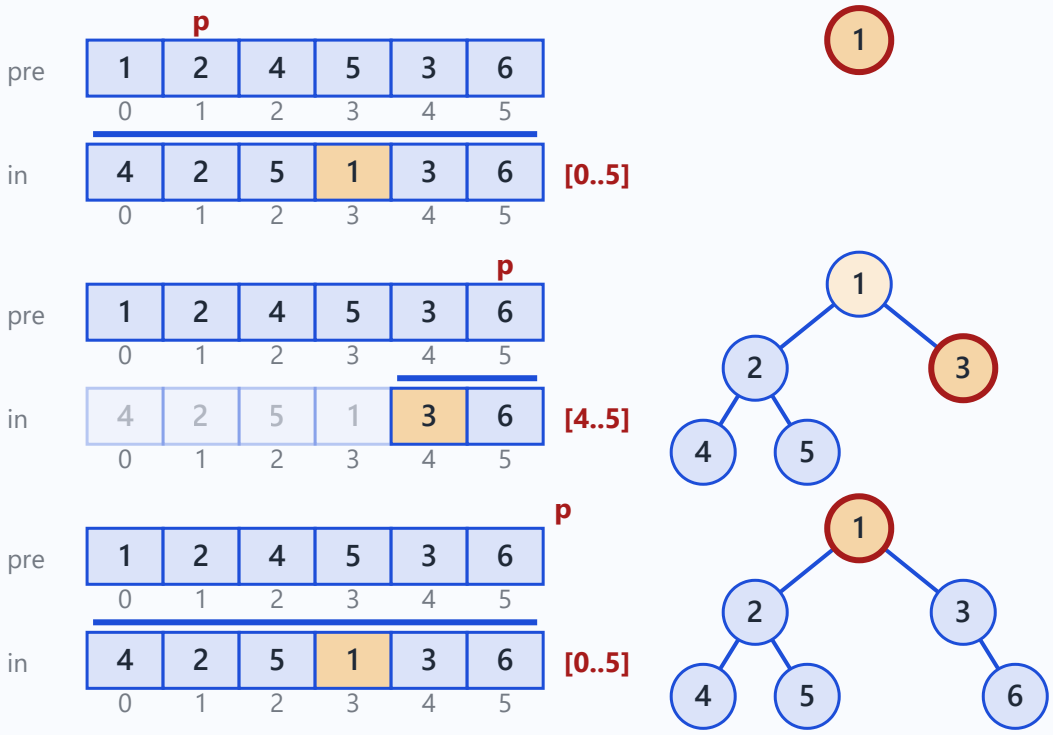


Figure 9.43: Three moments of `build`. Top: root 1 created and `in` split around it. Middle: left subtree (2, 4, 5) built; now `in[4..5]` for the right subtree's root 3. Bottom: done.

The linear search for `m`, one per node over its range, sets the cost: *O(n log n)* for a balanced tree, whose ranges halve per level, and $O(n^2)$ for a skewed one, whose ranges shrink by one. Storing each inorder value's index in a `HashMap` beforehand finds `m` in *O(1)*, bringing the build down to *O(n)*.

> **Common mistake**
>
> Building the left subtree first from postorder. Read backward, postorder gives the root, then the root of the *right* subtree, so read it from the end and build the right subtree first; otherwise right-subtree elements land in the left.

9.6.9 The cost of traversals

Every traversal visits each node once with constant work, so all take *O(n)* time. They differ in extra memory.

Method	Time	Extra memory	Why?
Recursive preorder / inorder / postorder	*O(n)*	*O(h)*	At most *h + 2* frames on the call stack (including the `null` call)
Preorder / inorder / postorder with a stack	*O(n)*	*O(h)*	The explicit stack takes the place of recursion

Method	Time	Extra memory	Why?
Level order with a queue	*O(n)*	*O(w)*	*w*: number of nodes on the widest level

Table 9.26: The cost of the traversal methods.

Depth-first memory follows height: small for a balanced tree ($h \approx log_2\ n$), *O(n)* for a skewed one ($h = n - 1$), where recursion may hit the stack limit and throw `StackOverflowError` while the stack-based version, using heap memory, keeps going.

Level-order memory follows width. A perfect tree's last level of about *n/2* nodes costs the queue *O(n)*; a skewed tree never queues more than one or two. So the families are opposites: level order is frugal on deep, narrow trees, depth-first on wide, shallow ones — on a perfect tree, *O(n)* versus *O(log n)* for recursive inorder.

9.7 Common mistakes

Most tree bugs fall into a few types. The table gathers the mistakes discussed throughout; use it as a checklist when reviewing tree code.

Mistake	What happens	The fix
Forgetting the base case	Without a `node == null` check before accessing `node.left`, a `NullPointerException`.	Every recursive tree method starts with the empty-tree case.
Mixing height conventions	Every value is off by one.	In edges: leaf 0, empty tree −1. In nodes: leaf 1, empty tree 0. Pick one and use it consistently.
Treating full and complete as the same	Wrong classification.	Full: 0 or 2 children at every node. Complete: all levels but the last filled, the last filling from the left. Neither implies the other.
Building a tree from one traversal	Several trees give the same output.	Inorder plus preorder or postorder is needed. Preorder plus postorder is ambiguous when nodes have one child.
Mixing up the index formulas	Wrong child or parent; the root may point to itself.	1-based: *2i*, *2i + 1*, *⌊i/2⌋*. 0-based: *2i + 1*, *2i + 2*, *⌊(i − 1)/2⌋*.
Recursion on a deep tree	In a skewed tree $h = n - 1$; hundreds of thousands of frames cause `StackOverflowError`.	A stack- or queue-based version.
Push order in stack-based preorder	Pushing left first gives root, right, left.	Push right first, then left.
Wrong base value	Returning 0 as the maximum of an empty tree gives wrong results for negative values.	The base value must be the identity of the combining operation (`Integer.MIN_VALUE`, 0, 1).

Table 9.27: Common mistakes with trees.

These mistakes share a common root: failing to mirror the tree's recursive structure in the code. A tree is either empty or a root with two subtrees, and a correct tree method handles both cases explicitly — returning a meaningful value for the empty tree, and for a nonempty one combining the subtrees' results with attention to where the root goes. That last point is also the only difference among the traversals: does the root come before, between, or after its subtrees?

9.8 Summary

- A **tree** is a hierarchical structure of nodes and of edges joining them in parent–child relationships. There is a single root, and every other node has exactly one parent. Hence there are no cycles, a tree of n nodes has $n - 1$ edges, and there is exactly one path between any two nodes.
- An **internal node** has at least one child; a **leaf** has none. Parent, child, sibling, ancestor, and descendant are used as in a family tree; being on the same level does not make two nodes siblings.
- **Degree** is the number of children. **Depth** is the number of edges from the root to a node, **height** the number from a node to its farthest leaf. The height of a tree is that of its root, the largest depth. Sources that count nodes give every value one larger.
- In a **binary tree** every node has at most two children, and left is distinguished from right. **Full**: 0 or 2 children at every node. **Complete**: all levels but the last filled, the last filling from the left. **Perfect**: all leaves at the same depth; both full and complete.
- A binary tree has at most 2^d nodes at depth d and at most $2^{h+1} - 1$ nodes for height h; $\lfloor \log_2 n \rfloor \le h \le n - 1$. A full binary tree has one more leaf than internal nodes.
- In the **array representation**, the 1-based index i has children $2i$ and $2i + 1$ and parent $\lfloor i/2 \rfloor$; it is ideal for complete trees (heaps) and wastes up to $2^n - 1$ cells on skewed ones. In the **linked representation** every node carries left and right references, $n + 1$ of which are `null`. General trees are stored with child lists or in first-child – next-sibling form.
- **Preorder** visits root-left-right, **inorder** left-root-right, **postorder** left-right-root; all three are different moments of the same Euler tour. **Level order** scans the tree level by level with a queue.
- Recursive traversals use at most $h + 2$ frames on the call stack. The explicit-stack versions do the same work in heap memory: preorder pushes right before left; inorder goes left and prints as it pops; postorder uses `peek` to check whether the right subtree is finished.
- On an expression tree, preorder gives prefix, postorder postfix, and parenthesized inorder infix notation; the value is computed in postorder. `size`, `height`, `countLeaves`, and `findMax` follow one pattern: a base value for the empty tree, then a combination of the subtrees' results.
- Inorder together with preorder (or postorder) determines a binary tree with distinct values uniquely. All traversals take $O(n)$ time; extra memory is $O(h)$ for the depth-first traversals and $O(w)$ for level order.

Topic	Key points
Terms	root, parent/child, leaf, internal node; n nodes → $n - 1$ edges

Topic	Key points
Measures	degree = number of children; depth = edges from the root; height = longest downward path
Binary tree	full: 0/2 children · complete: last level fills from the left · perfect: every level filled
Representation	array: index relations · linked: left/right references
Traversal	preorder root-L-R · inorder L-root-R · postorder L-R-root · level order with a queue
Cost	all traversals $O(n)$; extra memory $O(h)$ (stack) or $O(w)$ (queue)

Table 9.28: Chapter summary.

9.9 Key terms

Each term links to the course glossary, which gives its Turkish equivalent and its use in other chapters.

tree · node · edge · root · parent · child · sibling · ancestor · descendant · leaf · internal node · degree · depth · level · height · path · subtree · cycle · position · abstract data type (ADT) · binary tree · full binary tree · complete binary tree · perfect binary tree · binary search tree (BST) · heap · traversal · preorder · inorder · postorder · level-order · breadth-first search (BFS) · depth-first search (DFS) · expression tree · recursion · base case · call stack · stack frame · queue

9.10 Test Yourself

1. In a general tree, A is the root; A's children are B and C, B's children are D, E, F, C's child is G, and G's children are H and I. Which nodes are leaves and which internal? What are the degree, the number of edges, and the height of the tree? What are the depth and height of G?

   Answer

   The leaves are D, E, F, H, I and the internal nodes A, B, C, G (5 + 4 = 9 nodes). B has the largest degree (3), so the degree of the tree is 3. There are $9 - 1 = 8$ edges. The longest path is A→C→G→H, so the tree has height 3. G has depth 2 (A→C→G) and height 1 (G→H).

2. What are the smallest and largest possible numbers of nodes in a binary tree of height 5? What is the smallest possible height of a binary tree with 100 nodes?

   Answer

   At least $h + 1 = 6$ (a chain) and at most $2^6 - 1 = 63$ (a perfect tree). For 100 nodes, height 5 is not enough (at most 63 nodes) but 6 is (at most 127); the smallest height is $\lceil \log_2 100 \rceil = 6$.

3. How many leaves and internal nodes does a full binary tree with 21 nodes have? Stored in the linked representation, how many `null` child references does it have?

   Answer

   From $n = 2i + 1$, $i = 10$ and $l = i + 1 = 11$. The number of `null` references is $n + 1 = 22$. Check: each leaf has two `null` references, $2 \cdot 11 = 22$, and since both children of every internal node are present, there are no others.

4. In the 0-based array representation, what are the indices of the parent and children of the node at index 9? At which index does the same node sit in a 1-based array, and what are the corresponding indices there?

   Answer

   0-based: parent $\lfloor (9 - 1)/2 \rfloor = 4$, left child $2 \cdot 9 + 1 = 19$, right child 20. In a 1-based array the same node is at index 10: parent $\lfloor 10/2 \rfloor = 5$, children 20 and 21. Every 1-based index is exactly one more than its 0-based counterpart.

5. The root is 10; 10's children are 5 and 15; 5's children are 2 and 7; 7 has only a left child, 6; 15 has only a right child, 20. What are the preorder, inorder, postorder, and level-order outputs? Is this tree a binary search tree?

   Answer

   Preorder: `10 5 2 7 6 15 20`. Inorder: `2 5 6 7 10 15 20`. Postorder: `2 6 7 5 20 15 10`. Level order: `10 5 15 2 7 20 6`. The inorder output is in increasing order, so the tree is a binary search tree.

6. `preorderIterative` runs on the tree of the previous question. What is on the stack (bottom to top) at the end of each pass? What is the most elements the stack holds?

   Answer

   Initially `[10]`. Pop 10, push 15 and 5: `[15, 5]`. Pop 5, push 7 and 2: `[15, 7, 2]`. Pop 2: `[15, 7]`. Pop 7, push 6: `[15, 6]`. Pop 6: `[15]`. Pop 15, push 20: `[20]`. Pop 20: `[]`. The stack holds at most 3 elements, and the output is in preorder.

7. Which operator is at the root of the tree for $((9 - 3) / (1 + 2)) \times 4$? What are its prefix and postfix

forms, and what value does `evaluate` return?

Answer

The last operation performed is the multiplication, so the root is `×`; the root of its left subtree is `/` and its right child is 4. Prefix: `× / − 9 3 + 1 2 4`. Postfix: `9 3 − 1 2 + / 4 ×`. `evaluate` computes *9 − 3 = 6*, then *1 + 2 = 3*, *6 / 3 = 2*, and finally *2 × 4 = 8*, returning 8.

8. Which tree has inorder output `8 4 9 2 5 1 3 7` and postorder output `8 9 4 5 2 7 3 1`? What is its preorder output?

Answer

The last element of the postorder output is the root: 1. In the inorder output, 1 has `8 4 9 2 5` on its left and `3 7` on its right. In the right subtree, the next-to-last postorder element, 3, is the root, with 7 to its right. In the left subtree the root is 2, with `8 4 9` on its left and 5 on its right. The root of `8 4 9` is 4, with 8 as left and 9 as right child. The tree is 1(2(4(8, 9), 5), 3(—, 7)). Preorder: `1 2 4 8 9 5 3 7`.

9. How many different binary trees have preorder output `A B` and postorder output `B A`? What additional information determines the tree?

Answer

Two: B can be either the left or the right child of A, and both trees have the same preorder and postorder outputs. The inorder output tells them apart: `B A` if B is the left child, `A B` if it is the right. The ambiguity arises because A has only one child.

10. Suppose the base case of `height` is written `return 0` instead of `return -1`. What does the method return for the four-node tree rooted at 9 (9's children are 2 and 3; 2's left child is 4)? For a leaf?

Answer

A leaf returns *1 + max(0, 0) = 1*. Node 2 returns *1 + max(1, 0) = 2*, node 3 returns 1, and the root *1 + max(2, 1) = 3*. The method now measures height in nodes: every value is one more than under the edge-based definition (the correct answer is 2).

9.11 Further reading

- M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, *Data Structures and Algorithms in Java* (6th ed., 2014), Chapter 8: defines the general and binary tree ADTs through the position abstraction and gives linked and array implementations; covers traversals, the Euler tour, and expression trees

with Java code close to that of these notes.

- D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms* (3rd ed., 1997), Section 2.3: the classic treatment of tree traversal, covering nonrecursive traversal algorithms, threaded binary trees, and the representation of general trees by binary trees in detail.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms* (4th ed., 2022), the section on representing rooted trees in Chapter 10 and Appendix B.5: the first-child – next-sibling representation and the definitions and counting results for trees, in a more formal style.

9.12 References

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022.
2. M. T. Goodrich, R. Tamassia, M. H. Goldwasser, *Data Structures and Algorithms in Java*, 6th ed., Wiley, 2014.
3. D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, 3rd ed., Addison-Wesley, 1997.
4. Oracle, *Java SE 17 API Specification*: documentation for `java.util.Deque`, `java.util.ArrayDeque`, `java.util.Queue`, and `java.util.LinkedList`.
5. R. Sedgewick, K. Wayne, *Algorithms*, 4th ed., Addison-Wesley, 2011.

Search Trees

Binary search in a sorted array is fast, but inserting means shifting everything after the new element. A linked list makes insertion easy but forces a search to walk the whole list. Search trees combine the strengths of both. Elements are placed by key so that at every node “left or right?” has exactly one answer, and search, insertion and deletion follow a single path down from the root, at a cost set by the tree's height. We start with the binary search tree (BST) and its operations, see why height decides everything, and then study AVL, red-black and splay trees, which keep the height small by themselves. The last part covers B-trees and B+ trees, built for large data on disk.

Code for this chapter: `github.com/sercankulcu/data-structures-java/Bolum10/src`

LEARNING OUTCOMES

BST operations
Implement search, insertion and deletion in a binary search tree; answer successor, predecessor and range queries.

Why balance matters
Explain how tree height drives the cost of operations; show why the same keys inserted in different orders give very different trees.

Balancing
Interpret AVL rotations and the red-black tree rules; work out, step by step, the balanced tree that results from a sequence of insertions.

Comparison
Choose among BSTs, AVL, red-black, splay and B-trees according to the intended use.

10.1 Search trees

Nobody looks up “Smith” in a phone book by starting at page one. You open it in the middle, see which half the name is in, and discard the other, which works only because the book is sorted. On a computer this is binary search: about $log_2 n$ comparisons among n sorted elements.

The weakness of a sorted array is updating: adding an entry shifts everything after it, n moves in the worst case. A linked list inserts and deletes cheaply but cannot jump to its middle, so a search steps through it one element at a time. Search trees sit in between. Being linked, they update by changing a few links; being ordered, they let each search step eliminate part of the candidates.

10.1.1 What is a search tree?

First, the vocabulary. A tree consists of nodes joined by edges. The topmost node is the **root**; nodes without children are **leaves**. The **depth** of a node is the number of edges from the root down to it, so the root has depth 0, and the **height** h of the tree is the depth of its deepest node. A single node has height 0; the empty tree, by convention, −1.

Each node of a search tree holds an element with a **key**, a comparable value that identifies it: a student ID, a dictionary word, a surname. The rest of the element (name, address, GPA) is stored alongside. To keep examples simple, our keys are mostly integers and nodes hold nothing else.

Definition 10.1

A **search tree** is a tree whose nodes hold elements kept in an order determined by their keys, arranged so that a search for a key needs to descend into only one subtree at each node. A search therefore starts at the root, chooses one branch per level, and visits at most $h + 1$ nodes.

That last sentence is the whole chapter in brief. Search, insertion and deletion each follow one path down from the root, at most as long as the height, so they cost $O(h)$: fast while the tree stays short, slow when it grows tall.

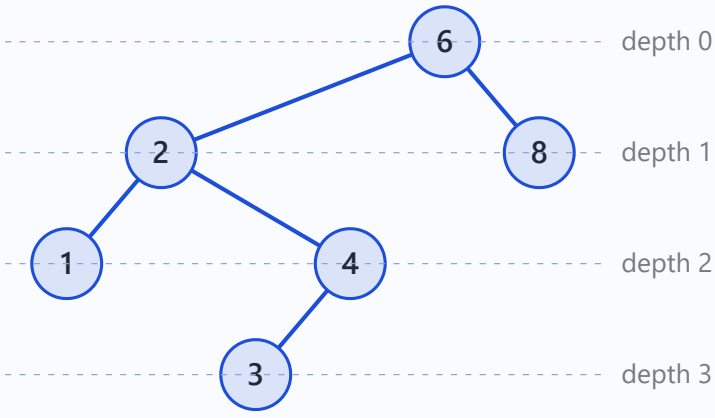


Figure 10.1: A six-node search tree rooted at 6. Depths: 6 is at 0, 2 and 8 at 1, 1 and 4 at 2, 3 at 3. The deepest node is 3, so the height is $h = 3$.

How tall can an n-node tree be? At worst every node has one child, the tree is a chain, and the height is $n - 1$. At best every level is full. Depth d holds at most 2^d nodes, so a binary tree of height h has at most $1 + 2 + 4 + \ldots + 2^h = 2^{h+1} - 1$ nodes, and $n \le 2^{h+1} - 1$ gives $h \ge \log_2(n + 1) - 1$: about $\log_2 n$.

Example 10.1

Store a million keys ($n = 10^6$) in a search tree. Perfectly balanced, it has height $\log_2 10^6 \approx 19.93$, so a search makes about 20 comparisons. Degenerated into a chain, it has height 999,999, and the worst search makes a million. Same keys, same operation, different shape: a ratio of about 50,000.

This gap is why the rest of the chapter exists. A plain BST promises nothing about height. AVL and red-black trees repair themselves locally after each update and guarantee $h = O(\log n)$. A splay tree meets that bound on average over long sequences, and a B-tree packs many keys per node, making the tree short and wide.

Common mistake

Counting height in nodes. Height counts *edges* from the root to the deepest leaf: the path 6 → 2 → 4 → 3 in Figure 10.1 has four nodes but three edges, so the height is 3. Some books count nodes, which shifts every formula by one. Here height always counts edges, and the empty tree has height −1.

10.1.2 Basic operations

A search tree offers what any sorted collection does. Table 10.1 lists the operations and our method names.

Operation	What it does	Example
Search	Returns `true` if the given key is in the tree, `false` otherwise.	`search(root, 5)` → `true`
Insertion	Adds a new element at the right leaf position without breaking the order.	`insert(root, 7)`
Deletion	Removes the element with the given key and preserves the order.	`delete(root, 40)`
Traversal	Visits all elements in a fixed order. Inorder traversal yields the keys in sorted order.	`inorder(root)` → 1 2 3 4 6 8

Table 10.1: Basic search tree operations; examples use the trees in the slides.

The order supports further queries: the minimum and maximum key; the successor of a key (the smallest larger key) and its predecessor (the largest smaller key); and all keys within a range. Each follows a single path from the root, so everything except traversal takes $O(h)$; traversal visits every

node once, in *O(n)*.

A hash table also searches and inserts in constant average time; ordered queries are what set a search tree apart. Because a hash table scatters its keys, questions like "the smallest key above 50" or "the keys from 20 to 45, in order" force it to scan everything. A search tree answers them in *O(h)* or *O(h + s)*, where *s* is the number of keys returned.

10.1.3 Kinds of search trees

We cover four kinds of binary search tree and two multiway trees. The binary kinds share one ordering rule and differ in how, and how tightly, they control height. Figure 10.2 shows a small example of each.

- **Binary search tree** (BST): each key is greater than every key in its left subtree and smaller than every key in its right subtree. No balance guarantee; the shape depends on insertion order.
- **AVL tree**: at every node the subtree heights differ by at most 1 (the **balance factor**); rotations restore the condition when it breaks.
- **Red-black tree**: nodes are red or black, and color rules bound the height by $2\ log_2(n + 1)$.
- **Splay tree**: rotations move each accessed node to the root. One operation can be costly, but the **amortized** cost is *O(log n)*.

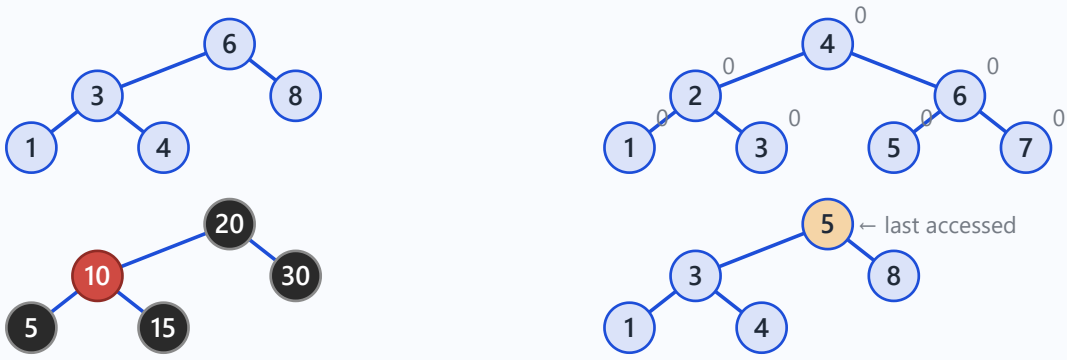


Figure 10.2: Four kinds of binary search tree. (a) BST: root 6; 3, 1, 4 on the left, 8 on the right. (b) AVL: a perfectly balanced seven-node tree, balance factor 0 everywhere. (c) Red-black: black root 20, red 10, black 30, 5 and 15. (d) Splay: the last key accessed, 5, is at the root.

In multiway trees (B-trees and B+ trees) a node holds several keys and more than two children. Designed mainly for data on disk, they are the subject of Section 10.7.

10.2 Binary search trees

In a binary tree each node has at most two children, left and right. A binary search tree adds an ordering rule that makes it behave like a "linked" sorted array.

10.2.1 The BST property

Definition 10.2

A **binary search tree** (BST) is a binary tree in which, for every node x, all keys in the left subtree of x are smaller than $x.key$ and all keys in its right subtree are larger. This is the **BST property**.

"All" matters: the condition relates a node to every descendant, not just its children. In Figure 10.3, the left tree is a valid BST; in the right one, 7 has been added as the right child of 4. That pair is fine, but 7 lies in the left subtree of 6 while being larger than 6, so the tree is not a BST.

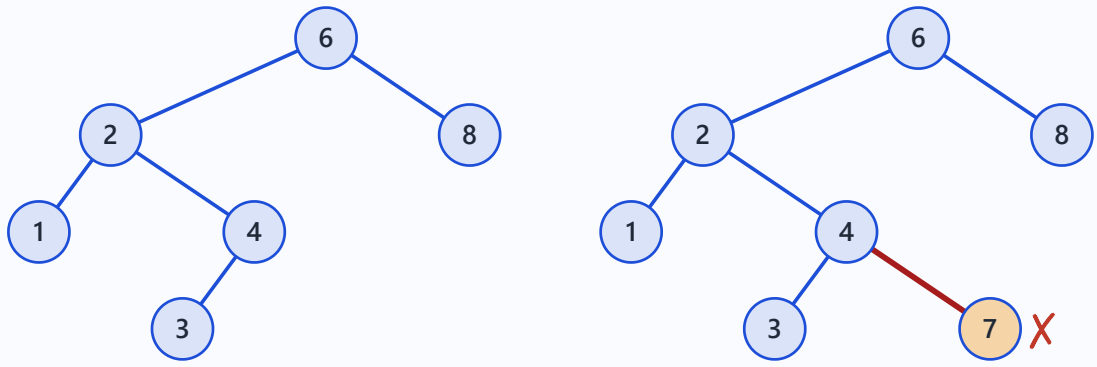


Figure 10.3: Left: a valid BST. Right: 7 added as the right child of 4. It is larger than its parent, as it should be, but also larger than 6 while in the left subtree of 6: not a BST.

So a node's key splits its subtree in two, smaller keys left and larger right, like the middle element of a sorted array. As binary search halves its range, a BST search knows at each node which side to take and discards the other.

Can a key appear twice? Not under this definition, with its strict inequalities. In practice you either reject duplicates, as our code does, or consistently send equal keys to one side, say the right. Search, insertion and deletion must follow the same rule, or an inserted key may never be found again.

Common mistake

Checking the BST property only between parents and children. "Every left child smaller, every right child larger" also holds for the right-hand tree in Figure 10.3, which is not a BST. The correct check carries lower and upper bounds down from the root (Section 10.3.12).

10.2.2 Node structure

A BST node has three fields: the data (the key) and references to its left and right children (Figure 10.4). A missing child is `null`, so a leaf has two `null` references. Compared with a linked-list node, there are two child references instead of one “next.”



Figure 10.4: The three fields of a binary tree node: left child reference, data and right child reference.

The slides write the node in two forms. The first holds only an `int` key:

```
class TreeNode {
    int data;
    TreeNode left, right;

    TreeNode(int data) {
        this.data = data;
        this.left = null;
        this.right = null;
    }
}
```

The constructor stores the data and sets both children to `null`, which Java does anyway; the assignments just show intent. The successor, range query and validation code use this class, with the short field names `left` and `right`.

The second form is a generic class. The key need not be an integer; any comparable type will do:

```
class BinaryTreeNode<E extends Comparable<E>> {
    E data;                    // element
    BinaryTreeNode<E> leftChild; // left child
    BinaryTreeNode<E> rightChild; // right child

    BinaryTreeNode(E data) {
        this.data = data;
        // children start out null
    }
}
```

The bound `E extends Comparable<E>` means two `E` objects can be compared with `compareTo`. `Integer`, `String` and `LocalDate` implement `Comparable`, so any of them can be a key. `a.compareTo(b)` is negative if $a < b$, 0 if equal, and positive if $a > b$; search, insertion and deletion look only at this sign. Here the fields are `leftChild` and `rightChild`.

Each node carries two references besides its data, so n nodes take $O(n)$ memory. Balanced trees add a constant per node, a height for AVL or a one-bit color for red-black, and total memory stays $O(n)$.

10.2.3 Minimum and maximum keys

The BST property tells us where the extremes are. A node with a left child cannot be the minimum, since something smaller is on its left. So the minimum is found by going left from the root until a node has no left child, and the maximum by going right (Figure 10.5).

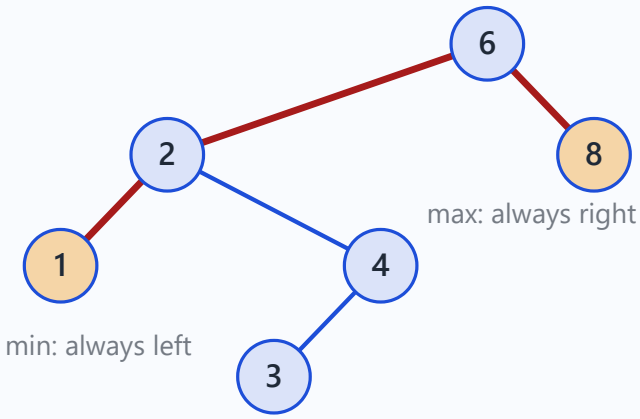


Figure 10.5: The minimum key 1 is reached by always going left from the root (6 → 2 → 1), the maximum key 8 by always going right (6 → 8).

The slides give both methods together:

```
BinaryTreeNode<E> minValue(
        BinaryTreeNode<E> node) {
    while (node.leftChild != null)
        node = node.leftChild; // go left
    return node;  // minimum
}
BinaryTreeNode<E> maxValue(
        BinaryTreeNode<E> node) {
    while (node.rightChild != null)
        node = node.rightChild; // go right
    return node;  // maximum
}
```

`minValue` starts at the given node and, while it has a left child, moves `node` there; when the left child is `null`, it returns the current node. `maxValue` does the same on the right. Both assume a nonempty tree: if `node` starts as `null`, `node.leftChild` throws a `NullPointerException`.

No keys are compared; the methods just follow links, one level per iteration, so they take at most h iterations, $O(h)$ time, and $O(1)$ extra memory.

Taking any node rather than the root is deliberate: deletion calls `minValue` on the right subtree of a two-child node to find its replacement. The same method serves for the whole tree or any subtree.

> **Common mistake**
>
> Writing the condition as `while (node != null)`. The loop then runs past the leftmost node to `null`, and the method always returns `null`. Stop one step earlier, at the node with no left child: check that a next node exists before moving.

10.2.4 Code trace: minValue and maxValue

We trace the tree of Figure 10.5, built by inserting 6, 2, 8, 1, 4, 3: root 6 with children 2 and 8; 2 with children 1 and 4; 4 with left child 3. Table 10.2 shows `node` and the condition in each iteration.

Call	Step	`node`	Condition	Result
`minValue`	1	6	`leftChild` = 2 ≠ null	go left
	2	2	`leftChild` = 1 ≠ null	go left
	3	1	`leftChild` = null	loop ends, `return` 1
`maxValue`	1	6	`rightChild` = 8 ≠ null	go right
	2	8	`rightChild` = null	loop ends, `return` 8

Table 10.2: Trace of `minValue(root)` and `maxValue(root)`.

Note that the deepest node, 3, is not the minimum: it is left of 4, which is right of 2, so 3 > 2. The minimum is in the *leftmost* node, not the deepest. Figure 10.6 shows three moments of `minValue` and the result of `maxValue`.

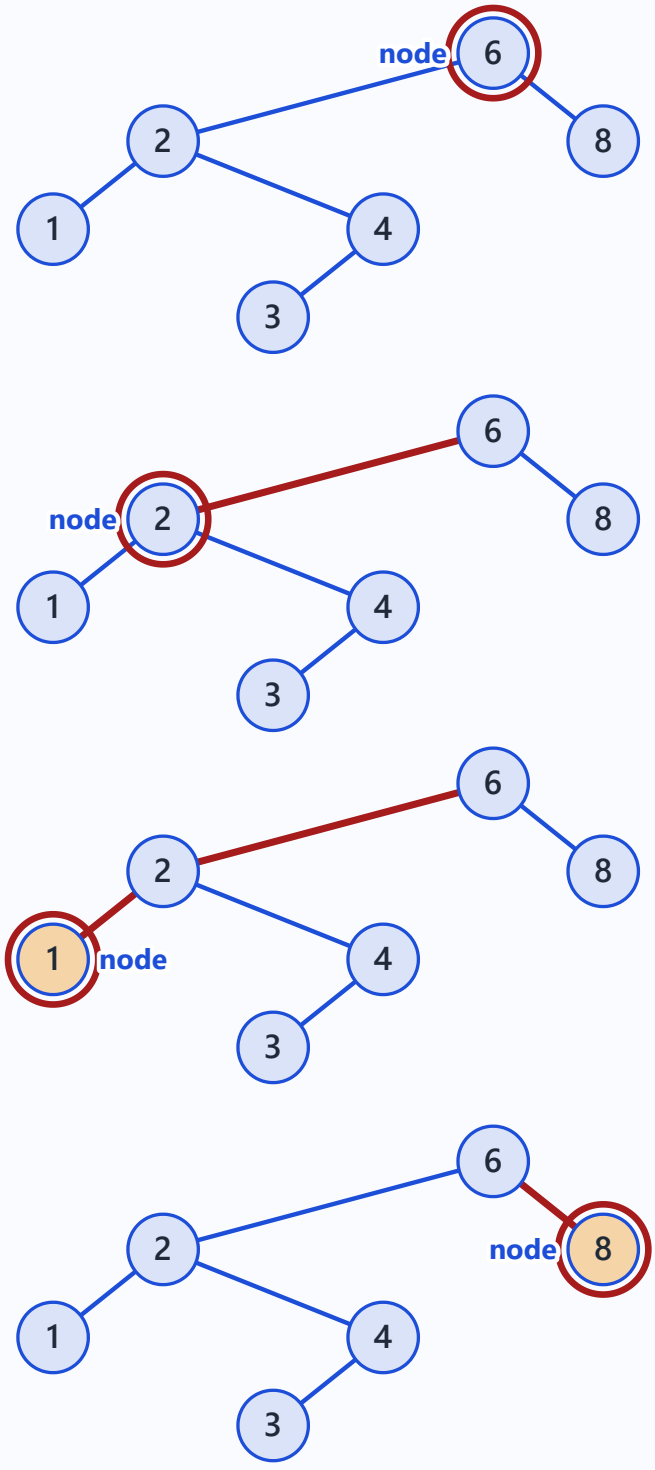


Figure 10.6: Four moments of the trace. (a) `minValue(root)`: `node` starts at 6. (b) `node = node.leftChild`: now at 2. (c) 1 has no left child; the loop ends with minimum 1. (d) `maxValue` moves from 6 to 8 and stops: maximum 8.

10.2.5 Inorder traversal

A traversal visits every node in a fixed order. Binary trees have three classic recursive orders, preorder, inorder and postorder, named for when the root is visited relative to its subtrees. Inorder visits the left subtree, then the root, then the right subtree.

```
void inorder(BinaryTreeNode<E> d) {
    if (d == null) return;
    inorder(d.leftChild);  // left
    System.out.print(d.data + " ");
    inorder(d.rightChild);  // right
}
```

On an empty subtree (`d == null`) the method returns at once: the base case. Otherwise its three lines traverse the left subtree, print the node's data, and traverse the right subtree.

In a BST this order prints the keys in increasing order. Take any node: its whole left subtree, all smaller keys, is printed before it, and its right subtree, all larger keys, after it. Every node thus comes after all smaller keys and before all larger ones, which is exactly sorted output.

Theorem 10.1 (Inorder traversal is sorted)

The inorder traversal of a binary search tree lists its keys in increasing order.

Proof. By induction on height. The empty tree gives empty output. For a tree of height h with root r, both subtrees are shorter, so by hypothesis each traversal lists its keys in increasing order. The output is "left list, r, right list," and by the BST property every left key is below r and every right key above it, so the whole list increases. □

The converse holds too: a binary tree whose inorder traversal is strictly increasing is a BST. So one way to check a tree is to traverse it inorder and test that the output increases.

Each node is printed once and makes two recursive calls, $O(n)$ calls in all (n nodes produce $n + 1$ empty calls), so the time is $O(n)$. The stack holds at most $h + 2$ frames, counting the empty call: $O(h)$ memory.

10.2.6 Code trace: inorder

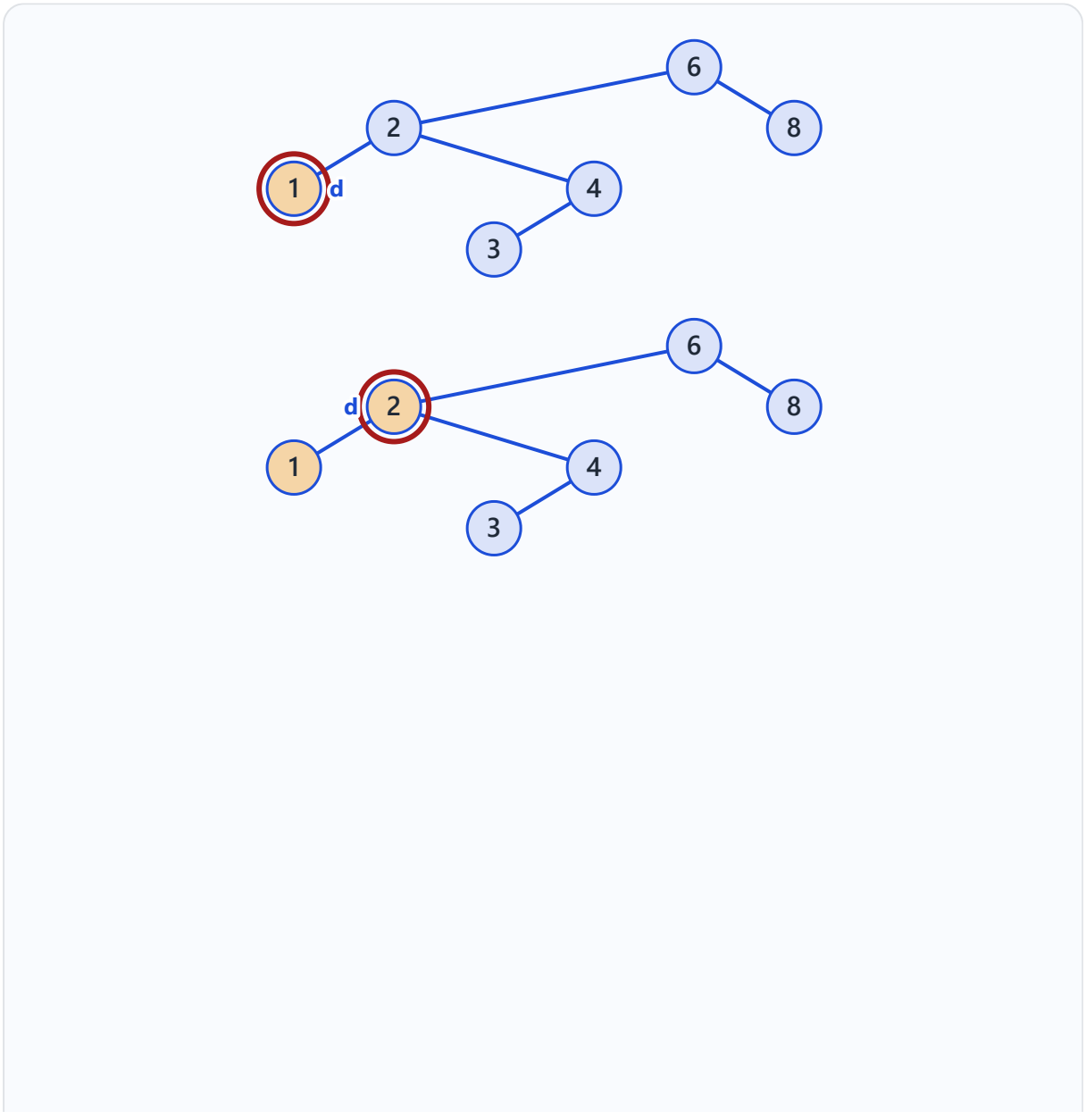
`inorder(root)` on the same tree produces the events of Table 10.3. "Pending" lists, bottom to top, the nodes on the call stack waiting for their left subtrees.

Step	Event	Pending	Output
1	`inorder(6)` → left first: `inorder(2)`	6	
2	`inorder(2)` → left first: `inorder(1)`	6, 2	

Step	Event	Pending	Output
3	`inorder(1)` : left is `null` , returns at once; print 1	6, 2	1
4	right of 1 is `null` ; frame closes; print 2	6	1 2
5	right of 2: `inorder(4)` → left first: `inorder(3)`	6, 4	1 2
6	`inorder(3)` : left and right are `null` ; print 3	6, 4	1 2 3
7	print 4; right of 4 is `null` ; `inorder(4)` and `inorder(2)` close	6	1 2 3 4
8	print 6; right subtree: `inorder(8)`	—	1 2 3 4 6
9	`inorder(8)` : left is `null` ; print 8; right is `null`	—	1 2 3 4 6 8

Table 10.3: Trace of `inorder(root)` (tree: 6(2(1, 4(3, —)), 8)).

The output 1 2 3 4 6 8 is sorted. 6 is called first but printed fifth, since its four smaller keys are all in its left subtree; 8 is called last and printed last. Figure 10.7 shows four moments.



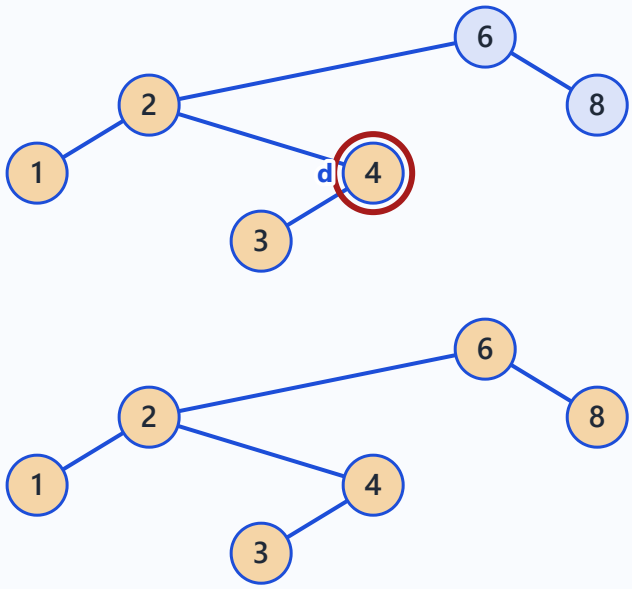


Figure 10.7: Four moments of the `inorder` trace; printed nodes are orange. (a) Left side done, 1 printed. (b) 2 printed. (c) 4 printed: output 1, 2, 3, 4. (d) All calls finished; output 1 2 3 4 6 8, each node printed once: *O(n)*.

Common mistake

Expecting sorted inorder output from any binary tree. That holds only for a BST; without the ordering rule, the order depends on the tree's shape. And in a BST, preorder and postorder are not sorted either. Only inorder is.

10.3 BST operations

All BST operations share one skeleton: start at the root, compare, go left if smaller and right if larger. They differ in where they stop and what follows. Search stops at an equal key or an empty link; insertion hangs a new node on the empty link; deletion removes the found node in one of three ways, depending on its children.

10.3.1 Search

Search follows three rules:

1. Start at the root. If the node's key equals the target, the search succeeds.
2. If the target is smaller, repeat in the left subtree; if larger, in the right subtree.
3. If you reach a `null` link, the key is not in the tree.

The third rule follows from the BST property: a key in the tree must lie in the subtree chosen at each step, since the other subtree's keys are all smaller or all larger. If that subtree is empty, the key is nowhere.

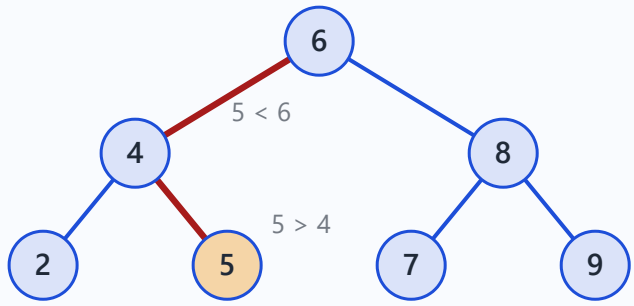


Figure 10.8: Searching for 5 in the tree with keys 6, 4, 8, 2, 5, 7, 9: left at 6 (5 < 6), right at 4 (5 > 4), and 5 is found.

The slides give a recursive search on the generic node class:

```
boolean search(BinaryTreeNode<E> root, E data) {
    if (root == null)
        return false;            // absent
    int k = data.compareTo(root.data);
    if (k == 0)
        return true;             // found
    else if (k < 0)
        return search(root.leftChild, data);
    else
        return search(root.rightChild, data);
}
```

The first `if` is the base case: an empty subtree has no keys, so the answer is `false`. Otherwise the target is compared once with the node's data, the result is kept in `k`, and the method returns `true` for `k == 0` or recurses left for `k < 0` and right for `k > 0`, passing the result up unchanged.

Comparing once and saving the result is clearer and cheaper: `compareTo` may be costly for a `String` or a complex object, and two `if`s would call it twice.

Common mistake

Comparing object keys with `==`, which tests whether two references are *the same object*, not whether values are equal. With `Integer` keys, the 1000 in a node and the 1000 searched for are separately boxed, so `data == root.data` is false and the key is missed. Java caches −128

to 127, so the bug appears only with larger numbers. Use `compareTo` for order and `compareTo(...) == 0` or `equals` for equality.

Each call descends one level with constant work. A successful search examines one node more than the found node's depth; an unsuccessful one at most $h + 1$. Both are $O(h)$, using $O(h)$ stack. Since the recursive call comes last, a `while` loop assigning `root = root.leftChild` or `root = root.rightChild` does the same in $O(1)$ extra memory.

10.3.2 Code trace: search

We trace a successful `search(root, 5)` and an unsuccessful `search(root, 10)` on the tree of Figure 10.8, 6(4(2, 5), 8(7, 9)). Each call opens a new stack frame.

Call	`root`	`k`	Decision
`search(root, 5)`	6	−1 (5 < 6)	left subtree: `search(4, 5)`
	4	1 (5 > 4)	right subtree: `search(5, 5)`
	5	0	found, `return true`; the frames at 4 and 6 return `true` unchanged
`search(root, 10)`	6	1 (10 > 6)	right: `search(8, 10)`
	8	1 (10 > 8)	right: `search(9, 10)`
	9	1 (10 > 9)	right: `search(null, 10)`
	`null`	—	9 has no right child: `return false`; the frames at 9, 8 and 6 return `false` unchanged

Table 10.4: Trace of `search(root, 5)` and `search(root, 10)`.

The successful search compared three nodes, one more than the depth of 5. The unsuccessful one also examined three (6, 8, 9) before its fourth call hit an empty link. With height 2, no search can exceed $h + 1 = 3$ comparisons, and both reached that bound; an unsorted array of the same seven keys would need seven for a miss.

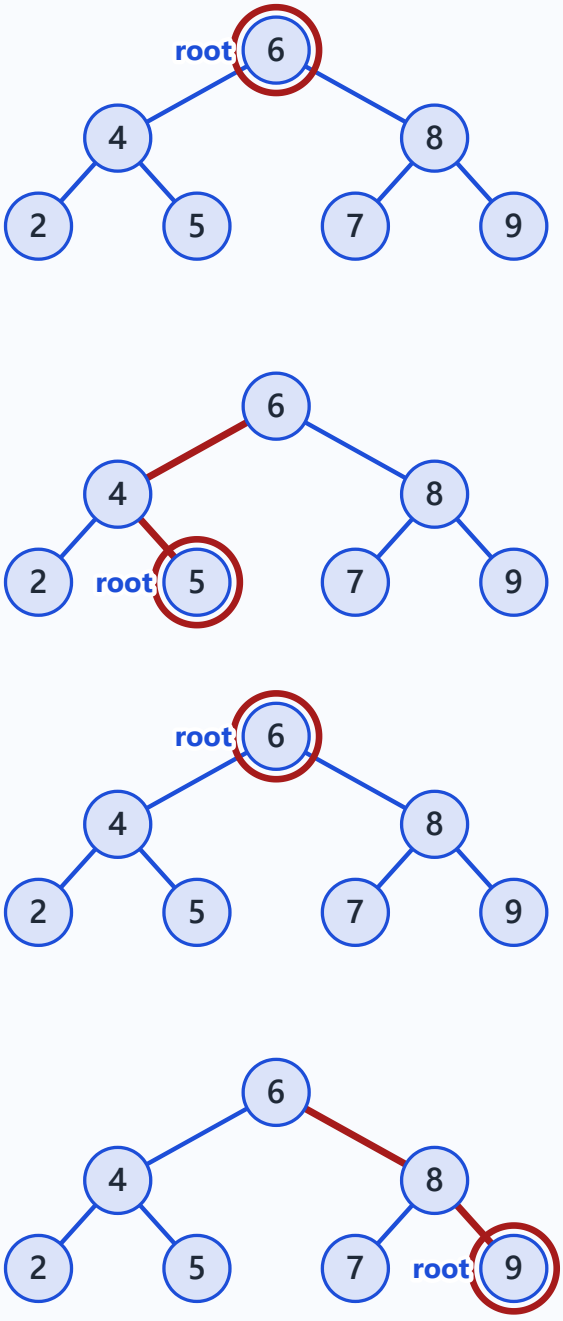


Figure 10.9: The search trace. (a) `search(root, 5)` : 5 < 6, `k = -1` . (b) Third frame: `root` is 5, `k = 0` . (c) `search(root, 10)` : 10 > 6, right. (d) `root == null` below 9: 10 is absent.

10.3.3 Insertion

A new key goes where a search for it ends: at a `null` link. Every node on the way sent the search to the correct side, so that link is the only place the key fits without breaking the BST property. The new node becomes a leaf there.

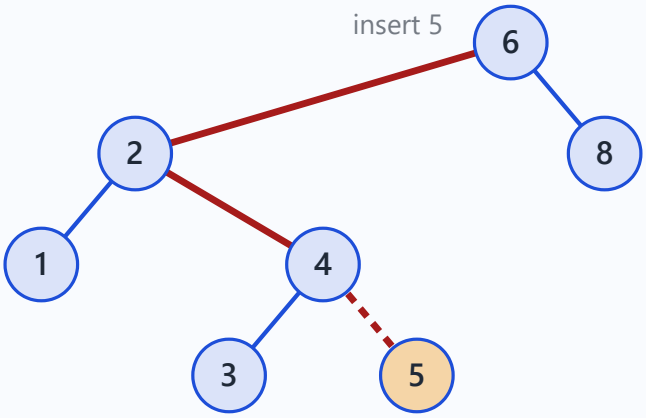


Figure 10.10: Inserting 5 into the tree with keys 6, 2, 8, 1, 4, 3: left at 6, right at 2, right at 4. The right link of 4 is empty, so 5 becomes a leaf there (dashed).

```
BinaryTreeNode<E> insert(
        BinaryTreeNode<E> root, E data) {
    if (root == null)
        return new BinaryTreeNode<>(data);
    int k = data.compareTo(root.data);
    if (k < 0)
        root.leftChild = insert(root.leftChild, data);
    else if (k > 0)
        root.rightChild = insert(root.rightChild, data);
    // k == 0: key already present, not inserted
    return root;
}
// usage: root = insert(root, 5);
```

The key feature is the **return value**: `insert` returns the root of its subtree after the insertion. For an empty subtree it creates and returns the new node, now that subtree's root. Otherwise the root stays the same: the method updates a child link with the recursive result and returns its own root.

The line `root.leftChild = insert(root.leftChild, data)` looks redundant, since the child is already there. Only the assignment at the bottom does real work: in the new node's parent, `root.rightChild` is still `null`, and the assignment links in the returned node. Elsewhere each link is rewritten with its own value. These “redundant” assignments save the code from tracking the parent.

The caller follows the same pattern, `root = insert(root, 5);`. For an empty tree, the first insertion creates the root, and only this assignment stores it.

Common mistake

Calling `insert(root, 8);` and ignoring the result. Arguments are passed by value, so the caller's `root` never changes: on an empty tree the new node is lost. On a nonempty tree the bug hides, since an inner `root.leftChild = …` links the leaf, until the root changes (empty tree, deleted root, rotation at the root) and the tree breaks.

An equal key (`k == 0`) skips both `if` s, so the method returns `root` unchanged and no duplicate is inserted. Insertion costs the same as search, *O(h)* time and *O(h)* stack. The new node is *always* a leaf and existing nodes never move, which is also why a plain BST's shape depends entirely on insertion order.

10.3.4 Code trace: insert

We insert 5 into the tree of Figure 10.10 with `root = insert(root, 5)`. Table 10.5 shows the descent and the return, with the value each frame assigns to its child link.

Step	Frame	`k`	Event
1	`root` = 6	−1	go left: `insert(root.leftChild, 5)`
2	`root` = 2	1	go right: `insert(root.rightChild, 5)`
3	`root` = 4	1	go right: `insert(root.rightChild, 5)`
4	`root` = null	—	`new` creates node 5 and returns it; no link points to it yet
5	`root` = 4	—	`root.rightChild = node(5)` : the new node replaces the empty link; `return` 4
6	`root` = 2	—	`root.rightChild = node(4)` : link unchanged; `return` 2
7	`root` = 6	—	`root.leftChild = node(2)` : link unchanged; `return` 6
8	`main`	—	`root = 6` : the root did not change

Table 10.5: Trace of `root = insert(root, 5)` (tree: 6(2(1, 4(3, —)), 8)).

The result is 6(2(1, 4(3, 5)), 8), with inorder 1 2 3 4 5 6 8. Three frames opened and closed, and only the right link of 4 really changed. Figure 10.11 shows the new node being created and linked.

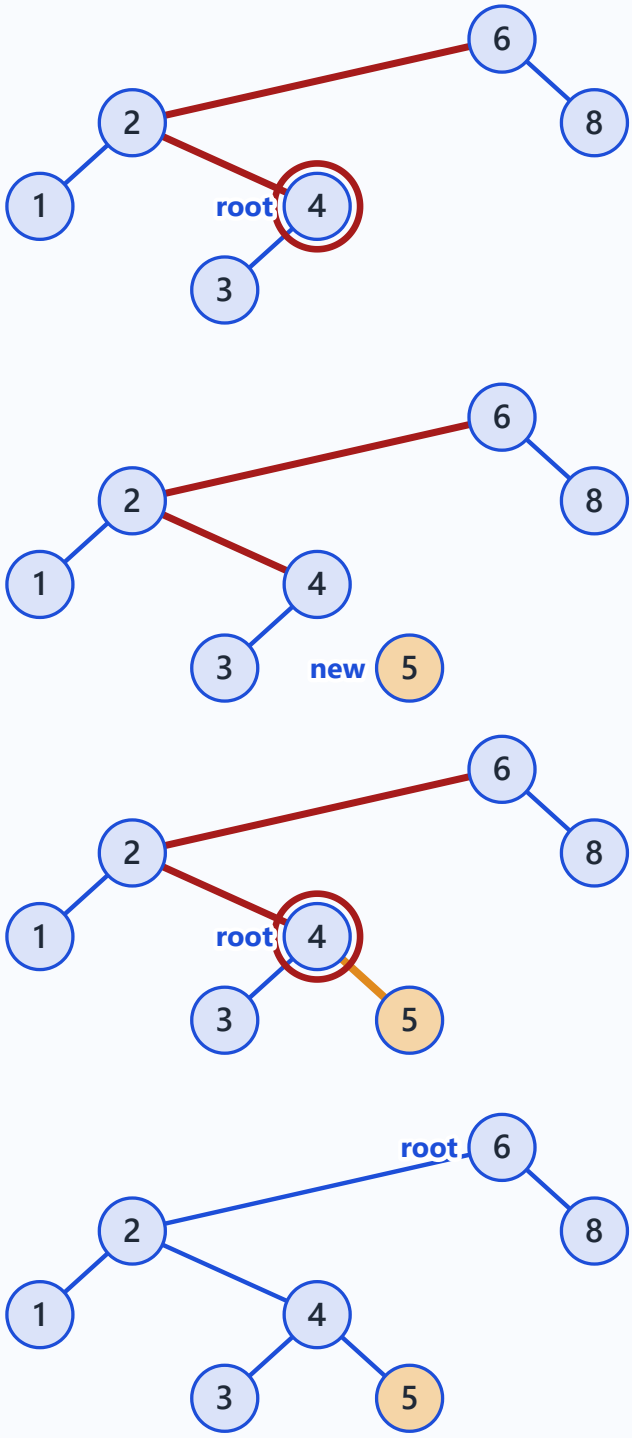


Figure 10.11: The insertion trace. (a) Right of 4, `root = null`. (b) Node 5 is created, unlinked. (c) The frame of 4 sets `root.rightChild` to it. (d) In `main`, `root = 6`: the root is unchanged; the new node is a leaf.

10.3.5 Successor, predecessor, ceiling and floor

Each element of a sorted list has a “next” and a “previous” neighbor; in a BST they are the successor and the predecessor.

Definition 10.3

The **successor** of a key x is the smallest key in the tree greater than x; its **predecessor** is the largest key smaller than x. x itself need not be in the tree. In inorder, they come right after and right before x.

For x in the tree there are two cases (Figure 10.12):

1. **With a right subtree**, the successor is its minimum: right once, then left all the way. The keys just above x are in that subtree, and the smallest of them is the successor.
2. **Without a right subtree**, it is the nearest ancestor that the root-to-x path passes on its *left*. The path to 5 is 6 → 4 → 5: it turns left at 6, so 5 < 6, and right at 4, so 4 < 5 cannot qualify. The successor is 6.

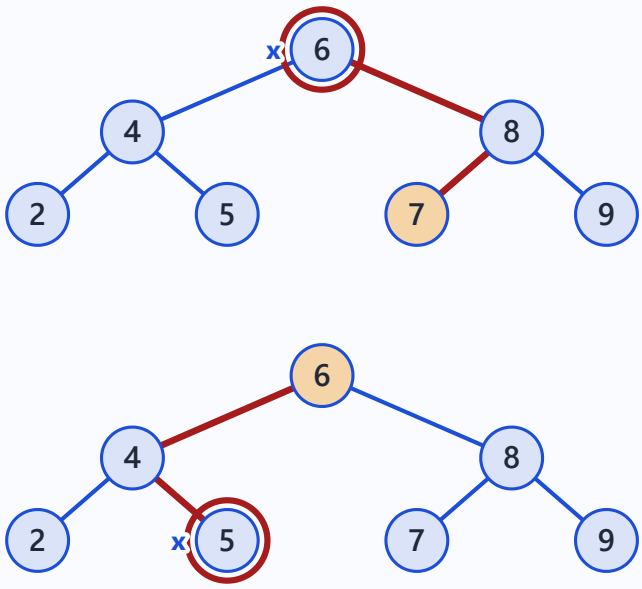


Figure 10.12: The two successor cases in 6(4(2, 5), 8(7, 9)). (a) 6 has a right subtree: right to 8, then left to 7; `successor(6) = 7`. (b) 5 has none: the nearest ancestor passed on its left is 6; `successor(5) = 6`.

The code merges both cases into one loop down from the root that keeps the “best candidate so far”:

```
TreeNode successor(TreeNode root, int x) {
    TreeNode candidate = null; // smallest key greater than x
    while (root != null) {
        if (x < root.data) {
            candidate = root;     // root is a candidate
            root = root.left;  // anything smaller is on the left
        } else
            root = root.right;  // keys greater than x are on the right
    }
    return candidate;
}
```

At each node, if $x < root.data$, the node is larger than x and a candidate; a better one, larger than x but smaller than this node, can only be on its left, so we record it and go left. If $x \geq root.data$, the node and its left subtree are all at most x and cannot qualify, so we go right. The loop ends at an empty link and returns the last candidate, or `null` if there was none, meaning no key exceeds x.

Both cases come out automatically. With a right subtree, the loop passes x, goes right once, then left while collecting candidates, ending at that subtree's minimum. Without one, the loop ends right after x, and the last candidate is the nearest ancestor passed on its left. The code does not even need x to be in the tree.

The successor has three relatives, listed in Table 10.6; Java's `TreeMap` and `TreeSet` provide all four.

Operation	Meaning	Java `TreeMap`	Example
successor	smallest key greater than x	`higherKey(x)`	`successor(5) = 6`
predecessor	largest key smaller than x	`lowerKey(x)`	`predecessor(6) = 5`
ceiling	smallest key $\geq x$	`ceilingKey(x)`	`ceiling(3) = 4`
floor	largest key $\leq x$	`floorKey(x)`	`floor(3) = 2`

Table 10.6: Ordered neighbor queries; the last column refers to the tree 6(4(2, 5), 8(7, 9)).

The predecessor mirrors the successor: if $x > root.data$, record the node and go right; otherwise go left. Ceiling differs from successor only on equality: for x in the tree, the ceiling is x itself, the successor the next key. Floor is likewise predecessor with equality allowed. Since 3 is absent from the example tree, ceiling(3) = 4 and floor(3) = 2. All four follow one path, in $O(h)$.

Example 10.2

In the same tree, find `floor(root, 7)` and `predecessor(root, 7)`.

1. `floor` : 7 ≥ 6, candidate 6; right (8). 7 < 8, left (7). 7 ≥ 7, candidate 7; right (`null`). Result 7: a key in the tree is its own floor.
2. `predecessor` : 7 > 6, candidate 6; right (8). Not 7 > 8, left (7). Not 7 > 7, left (`null`). Result 6, the largest key below 7.

Same path; the only difference is whether 7 itself counts as a candidate.

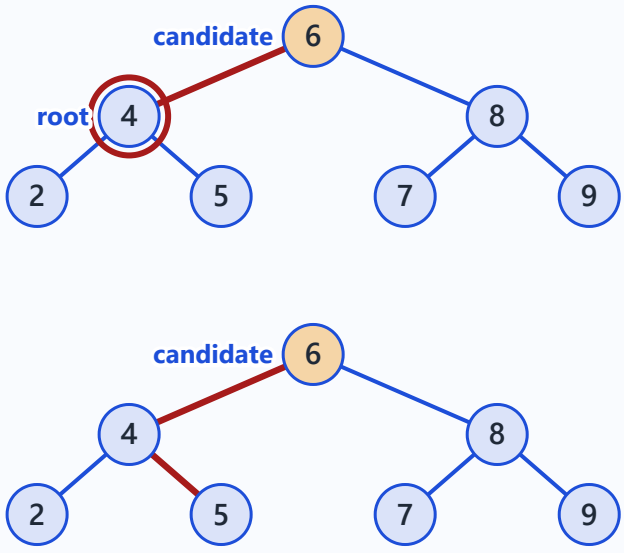
10.3.6 Code trace: successor

`successor(root, 5)` and `successor(root, 6)` illustrate the two cases. Each row of Table 10.7 is one loop iteration.

Call	`root`	Comparison	`candidate`	Direction
`successor(root, 5)`	6	5 < 6	6	left
	4	5 ≥ 4	6	right
	5	5 ≥ 5	6	right → `null`; `return` 6
`successor(root, 6)`	6	6 ≥ 6	null	right
	8	6 < 8	8	left
	7	6 < 7	7	left → `null`; `return` 7

Table 10.7: Trace of `successor(root, 5)` and `successor(root, 6)`.

In the first call the candidate, found at the root, never changes: 5 has no right subtree, so its successor is the ancestor 6. In the second it becomes 8, then 7, the minimum of the right subtree of 6. Each call examines three nodes. Figure 10.13 shows four moments.



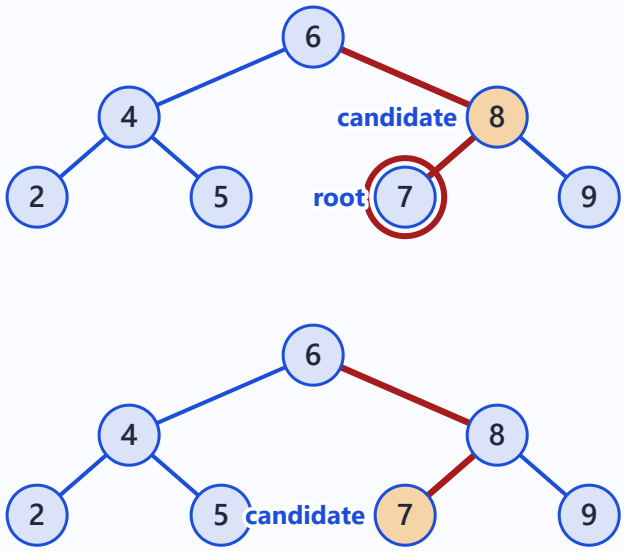


Figure 10.13: The `successor` trace. (a) 5 < 6: candidate 6, go left. (b) Loop done; `successor(5) = 6`. (c) `successor(root, 6)`: 6 < 8, candidate 8. (d) 7 is better; loop done, `successor(6) = 7`.

Common mistake

Taking the successor to be the right child. The right child of 6 is 8, but its successor is 7, the *minimum* of the right subtree. Conversely, a node without a right subtree may still have a successor: that of 5 is the ancestor 6.

10.3.7 Deletion: three cases

Deletion is the most delicate BST operation. A search finds the node in *O(h)*, which is then removed, according to its number of children, without breaking the BST property. The examples use the tree of Figure 10.14.

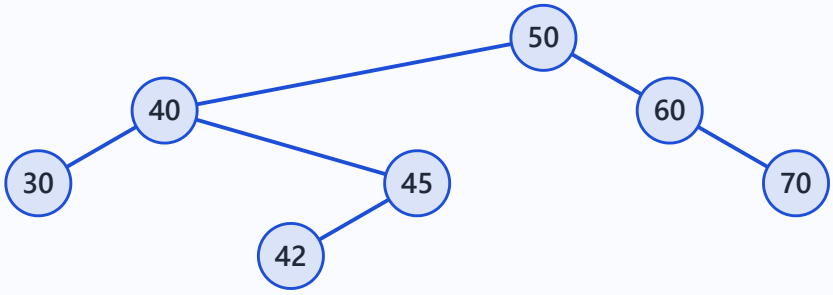
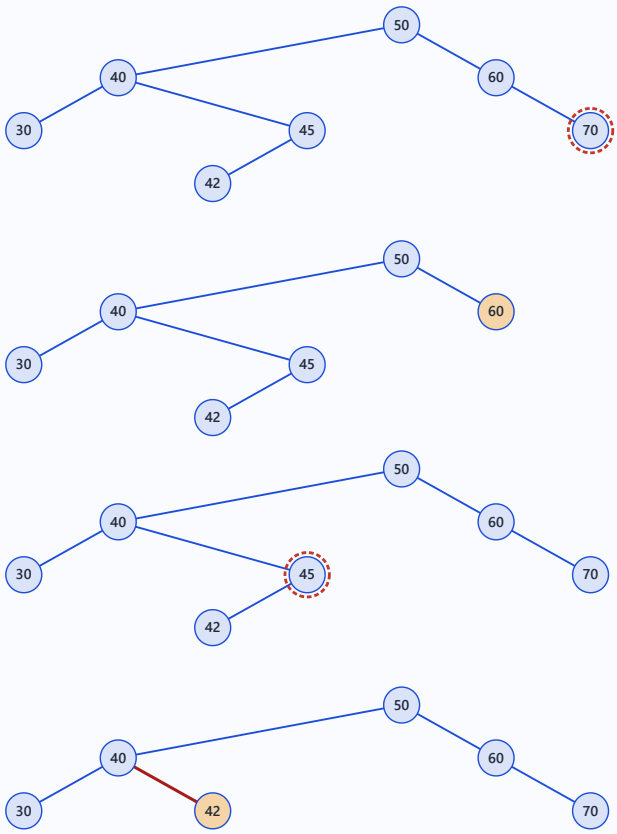


Figure 10.14: The deletion examples' tree, 50(40(30, 45(42, —)), 60(—, 70)). 70 and 30 are leaves, 45 and 60 have one child, 40 and 50 two.

1. **Leaf.** Set the parent's link to `null`. Deleting 70 nulls the right link of 60.

2. **One child.** Link the parent directly to the only child. Deleting 45 points 40 at 42; deleting 60 points 50 at 70.
3. **Two children.** The parent has one link to offer but the node has two children, so it cannot be removed directly. Instead, replace the node's *value* with that of its inorder successor (minimum of the right subtree) or predecessor (maximum of the left subtree), then delete that node from its own position. Example: deleting 40.

The one-child case clearly preserves the BST property. 45 and all its descendants lie in the right subtree of 40 and exceed 40, so linking 42 to the right of 40 changes nothing; 42 also stays in the left subtree of 50, below 50. The subtree simply moves up a level, its order intact.



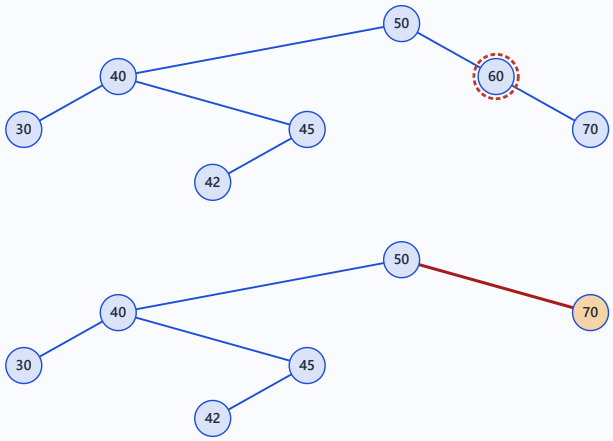
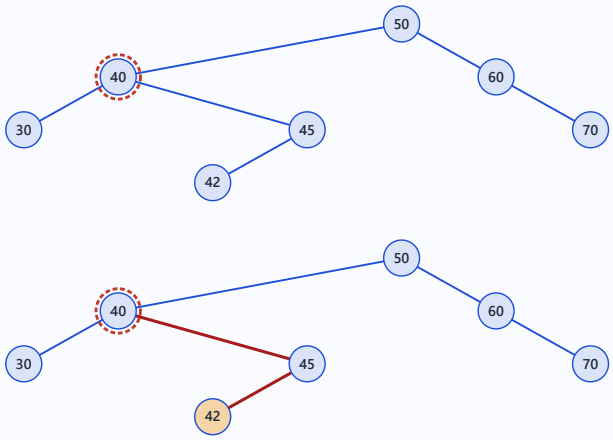


Figure 10.15: Deleting a leaf and one-child nodes, each from the original tree. (a)–(b) Leaf 70 goes; 60's right link becomes `null`. (c)–(d) 42, the only child of 45, replaces it under 40. (e)–(f) 70, the only child of 60, replaces it under 50.

Why the successor? The replacement must exceed everything in the left subtree and be smaller than everything in the right. The keys adjacent in inorder, predecessor and successor, both qualify. The successor of 40 is 42: as the minimum of the right subtree it is below the other keys there, and being above 40 it exceeds the whole left subtree. Putting 42 in place of 40 keeps the BST property.

Deleting the successor from its old position is always easy. It was found by going left in the right subtree until there was no left child, so it is a leaf or has only a right child: Case 1 or Case 2. The recursion never meets a two-child node again.



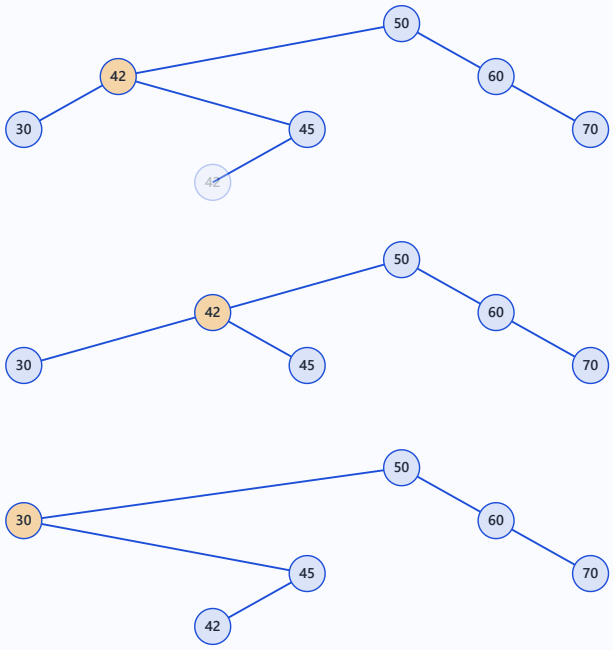


Figure 10.16: Deleting 40, which has two children. (a) 40 has children 30 and 45. (b) Inorder successor: to 45, then left to 42. (c) 42 is copied into the node of 40; the old 42 is a duplicate. (d) The old 42, a leaf, is deleted from the right subtree (Case 1). (e) The predecessor, 30, works too; either choice gives a valid BST.

10.3.8 The deletion code

With the same “return the subtree's new root” pattern as insertion, one recursive method handles all three cases:

```
BinaryTreeNode<E> delete(
        BinaryTreeNode<E> root, E data) {
    if (root == null) return null;   // absent
    int k = data.compareTo(root.data);
    if (k < 0)
        root.leftChild = delete(root.leftChild, data);
    else if (k > 0)
        root.rightChild = delete(root.rightChild, data);
    else if (root.leftChild == null)
        return root.rightChild;  // leaf/right child
    else if (root.rightChild == null)
        return root.leftChild;  // left child only
    else {                     // two children
        root.data = minValue(root.rightChild).data;
        root.rightChild = delete(root.rightChild, root.data);
    }
    return root;
}
```

The first half is a search. If the key is absent (`root == null`), it returns `null`, leaving the tree unchanged. For `k < 0` or `k > 0` it recurses into the corresponding subtree and assigns the result to the child link.

At `k == 0` the node is found, and three branches handle the cases:

- If `root.leftChild == null`, return `root.rightChild`. This covers two cases: for a leaf it returns `null`, emptying the parent's link; for a node with only a right child it returns that child, which the parent then links to directly.
- If `root.rightChild == null` (left child only), return the left child.
- With two children, `minValue(root.rightChild)` finds the successor, whose value is copied into the node; `delete(root.rightChild, root.data)` then removes the successor, since `root.data` now holds its value.

The node is detached when the returned value is assigned to the parent's link: in the leaf and one-child cases the method returns the node's replacement, and the reference to the deleted node disappears, leaving it for the garbage collector. Calls are written `root = delete(root, 40);`, since a deleted root can only be replaced through this assignment.

The descent costs *O(h)*. In the two-child case `minValue` and the second `delete` continue below the node, together within *O(h)*. Deletion takes *O(h)*.

> **Common mistake**
>
> Copying the successor's value into a two-child node but forgetting to delete the successor. Then 42 appears twice and the tree is no longer a BST. Equally wrong is deleting with the old key: `delete(root.rightChild, data)` looks for 40 in the right subtree and finds nothing. Use the copied value: `delete(root.rightChild, root.data)`.

10.3.9 Code trace: three deletions

All three start from the tree of Figure 10.14; Table 10.8 shows the comparisons and returned links.

Call	Descent (k)	At the node found	Link changed	Resulting tree
`delete(root, 70)`	50 (1), 60 (1), 70 (0)	no left child → `return root.rightChild = null`	right link of 60 is `null`	50(40(30, 45(42, —)), 60)
`delete(root, 45)`	50 (−1), 40 (1), 45 (0)	left child, no right → `return root.leftChild` = 42	right link of 40 is 42	50(40(30, 42), 60(—, 70))
`delete(root, 40)`	50 (−1), 40 (0)	two children: successor 42; `root.data = 42`; `delete(subtree at 45, 42)`: 45 (−1), 42 (0), leaf → `null`	left link of 45 is `null`; the node of 40 now holds 42	50(42(30, 45), 60(—, 70))

Table 10.8: Trace of `delete(root, 70)`, `delete(root, 45)` and `delete(root, 40)`.

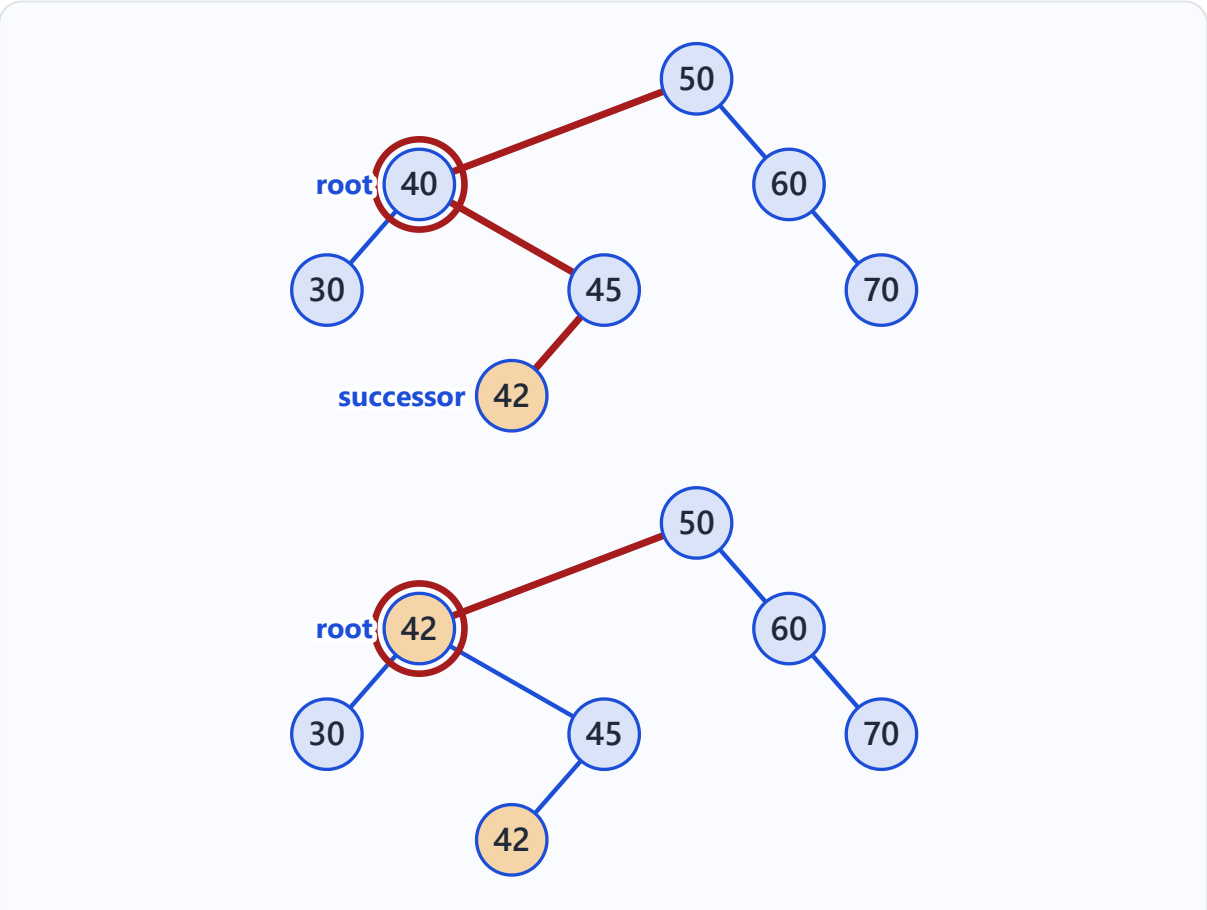
The inorder sequences of the results are 30 40 42 45 50 60, 30 40 42 50 60 70 and 30 42 45 50 60 70: the deleted key is gone, the rest in order. The root never changed; `root = delete(…)` in `main` wrote back 50 each time.

The third trace runs one deletion inside another; Table 10.9 shows it step by step.

Step	Frame	Event
1	`delete(50, 40)`	k = −1 (40 < 50): `root.leftChild = delete(root.leftChild, 40)`
2	`delete(40, 40)`	k = 0; two children (30 and 45)
3	`delete(40, 40)`	`minValue(45)` : left from 45 to 42; 42 has no left child → successor 42
4	`delete(40, 40)`	`root.data = 42` : the node now holds 42; the tree contains two 42s
5	`delete(45, 42)`	k = −1 (42 < 45): `root.leftChild = delete(root.leftChild, 42)`
6	`delete(42, 42)`	k = 0; no left child → `return root.rightChild = null`
7	`delete(45, 42)`	`root.leftChild = null` : the old 42 is detached; `return` 45
8	`delete(42, 40)`	`root.rightChild = node(45)` (link unchanged); `return` the node (value 42)
9	`delete(50, 40)`	`root.leftChild` = the same node; `return` 50; `main` : `root = 50`

Table 10.9: Details of the trace of `root = delete(root, 40)` .

The *node object* holding 40 never left the tree; only its value became 42. What was detached is the leaf holding the old 42. Figure 10.17 shows three moments and the result.



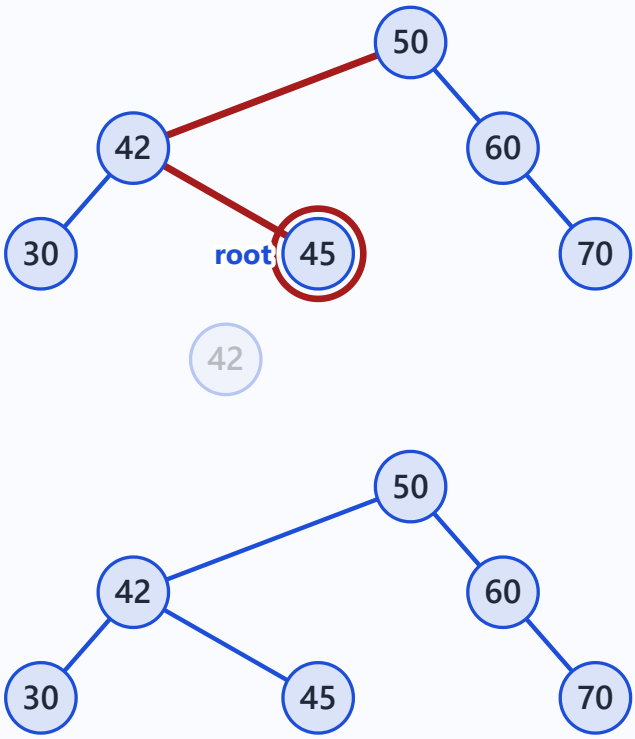


Figure 10.17: `delete(root, 40)`. (a) `minValue(root.rightChild)`: successor 42. (b) `root.data = 42`: 42 appears twice. (c) The link to the old 42 is now `null`. (d) Result 50(42(30, 45), 60(—, 70)): one node fewer, BST property intact.

Example 10.3

Starting again from the original tree, delete the root, 50.

1. `delete(50, 50)`: k = 0; two children (40 and 60).
2. `minValue(60)`: 60 has no left child, so it is its own successor.
3. `root.data = 60`; then `delete(subtree at 60, 60)`: k = 0, no left child → `return root.rightChild` = 70, a one-child deletion.
4. The root's right link now points to 70. Result: 60(40(30, 45(42, —)), 70); inorder 30 40 42 45 60 70.

Here the successor was the deleted node's right child and had a right child of its own: a successor need not be a leaf, but it never has a left child.

10.3.10 Analysis of BST operations

Every operation so far costs time proportional to the depth of the deepest node it reaches, at worst the height: *O(h)*. So the real question is how tall a BST with *n* keys gets.

That depends on insertion order. Inserting 4, 2, 6, 1, 3, 5, 7 gives a perfect tree of height 2. Inserting 1, 2, …, 7 puts each key right of the previous one, producing a chain of height 6. Such a **degenerate** tree is just a linked list that wastes memory.

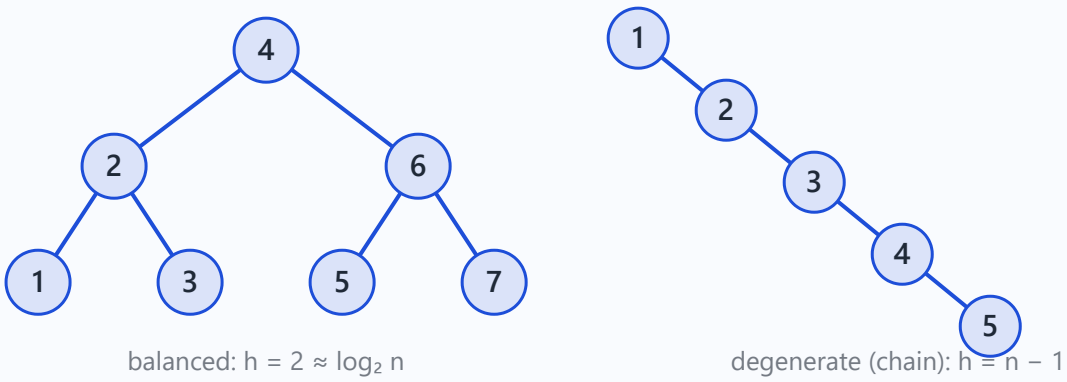


Figure 10.18: Left: a balanced 7-node tree of height 2 (about $\log_2 n$). Right: inserting 1, 2, 3, 4, 5 in order gives a chain of height $n - 1 = 4$.

Case	Height h	Search / insert / delete	$n = 10^6$
Balanced (best)	$\approx \log_2 n$	$O(\log n)$	≈ 20
Random order (average)	$\approx 2.99 \log_2 n$	$O(\log n)$	≈ 28 (average)
Degenerate (worst, sorted input)	$n - 1$	$O(n)$	10^6

Table 10.10: BST height and operation cost; the last column is the approximate number of steps for $n = 10^6$.

The middle row is encouraging. With random insertion order, the average depth is about $2 \ln n \approx 1.39 \log_2 n$, some 28 comparisons for a million keys, only about 40% above the best case. The expected *height* is larger, about $4.31 \ln n \approx 2.99 \log_2 n$, around 60 for a million keys; the table's height column shows this, its last column the average depth. The analysis matches quicksort's: each key, on becoming a root, splits the rest into two random groups.

Real data, however, is seldom random. Records from a file may be sorted already, student IDs grow with enrollment, timestamps only increase. Such input drives a plain BST to its worst case, with every operation $O(n)$, and inserting n sorted keys costs $1 + 2 + \ldots + (n - 1) = n(n - 1)/2 = O(n^2)$ comparisons: half a trillion for a million keys.

Common mistake

Saying "BST search is $O(\log n)$." Only for a balanced tree. A plain BST guarantees nothing about height; the correct bound is $O(h)$, with $h = n - 1$ in the worst case. $O(\log n)$ is guaranteed only by balanced trees such as AVL and red-black trees.

10.3.11 Range queries

A range query reports, in order, every key in *[lo, hi]*: "keys between 5 and 10," or "students with a GPA from 2.5 to 3.0." Traversing the whole tree inorder works but takes *O(n)* however few keys qualify. The BST property lets us do far better by never entering a subtree that cannot hold a key in range, a technique called **pruning**.

```
void range(TreeNode k, int lo, int hi) {
    if (k == null) return;
    if (lo < k.data)          // candidates on the left
        range(k.left, lo, hi);
    if (lo <= k.data && k.data <= hi)
        System.out.print(k.data + " ");
    if (k.data < hi)          // candidates on the right
        range(k.right, lo, hi);
}
```

This is inorder traversal pruned by two tests. All left-subtree keys are below `k.data`, so if `k.data ≤ lo` none can be in range and the left subtree is skipped; likewise, if `k.data ≥ hi`, the right subtree is skipped. The middle `if` prints the node when it is in range, and the left–node–right order keeps the output sorted.

Table 10.11 traces `range(root, 5, 10)` on 8(4(2, 6(5, 7)), 12(10, 14)).

Node	Left condition `5 < k.data`	In range?	Right condition `k.data < 10`	Output
8	true → go left	the node and its right side are handled after the left subtree		
4	false → 2 is pruned	no	true → go right	
6	true → go left	the node and its right side are handled after the left subtree		
5	false → left skipped	yes	true → right is `null`	5
6	back from the left; print 6; `range(7)`			5 6
7	true → left is `null`	yes	true → right is `null`	5 6 7
8	back from the left; print 8; `range(12)`			5 6 7 8
12	true → go left	no	false → 14 is pruned	
10	true → left is `null`	yes	false → right pruned	5 6 7 8 10

Table 10.11: Trace of `range(root, 5, 10)`.

The output 5 6 7 8 10 is sorted. The visited nodes are 8, 4, 6, 5, 7, 12, 10; 2 and 14 were never visited, because 4 was already below the lower bound and 12 above the upper. Figure 10.19 shows the start and end of the trace.

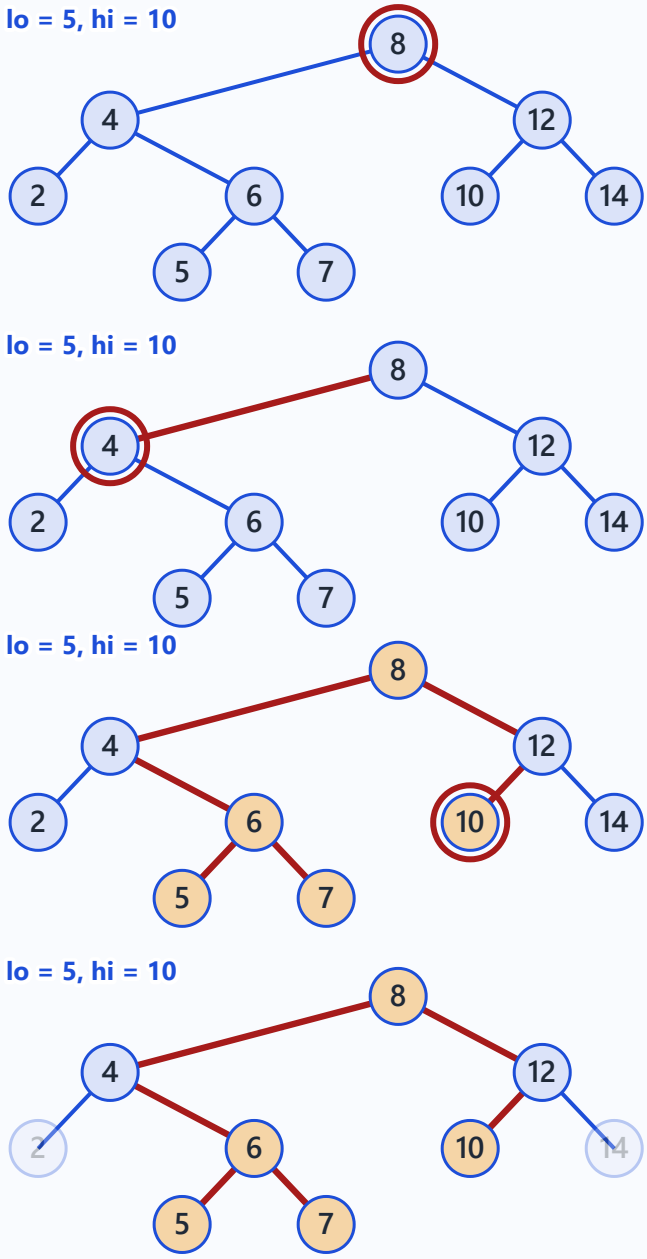


Figure 10.19: `range(root, 5, 10)`. (a) 5 < 8: go left. (b) 5 < 4 fails: the left subtree of 4 is pruned. (c) 12 ≥ 10: its right subtree is pruned. (d) Result 5 6 7 8 10; 2 and 14 (faded) are never visited.

For the cost, split the visited nodes into the s nodes in range, all printed, and the others, which lie on the two paths toward the ends of the range, each of at most $h + 1$ nodes. The total is $O(h + s)$: in a balanced tree of a million keys, a 10-key range takes a few dozen node visits.

10.3.12 Is a tree a BST?

To check whether a binary tree is a BST, remember that the property applies to whole subtrees: each node must agree with all its ancestors, not just its parent. Going left adds an upper bound, going right a lower bound, and every node must lie within all the bounds on its path from the root.

```
boolean isBST(TreeNode k, long min, long max) {
    if (k == null) return true;
    if (k.data <= min || k.data >= max)
        return false;          // out of range
    return isBST(k.left, min, k.data)
        && isBST(k.right, k.data, max);
}
// first call: isBST(root, Long.MIN_VALUE,
//                   Long.MAX_VALUE)
```

Each node receives an allowed interval *(min, max)*. The root's is unbounded, with `Long.MIN_VALUE` and `Long.MAX_VALUE` standing in for ±∞; since keys are `int`s, these `long` values never equal a key. A node outside its interval yields `false`; otherwise its value becomes the upper bound for its left child and the lower bound for its right, and the two results are joined with `&&`.

Table 10.12 checks the faulty tree on the right, 6(2(1, 4(3, 7)), 8).

Call	Allowed interval	Check	Result
`isBST(6)`	(−∞, ∞)	6 in range	left: `isBST(2, −∞, 6)`
`isBST(2)`	(−∞, 6)	2 in range	left: `isBST(1, −∞, 2)`
`isBST(1)`	(−∞, 2)	1 in range; children `null`	`true`
`isBST(4)`	(2, 6)	4 in range	left: `isBST(3, 2, 4)`
`isBST(3)`	(2, 4)	3 in range; children `null`	`true`
`isBST(7)`	(4, 6)	7 ≥ 6	`false`
`isBST(4)` = true && false; `isBST(2)` = true && false			`false`
`isBST(6)`: left is `false`; `&&` short-circuits and 8 is never checked			`false`

Table 10.12: Trace of `isBST(root, Long.MIN_VALUE, Long.MAX_VALUE)`.

The interval for 7 is *(4, 6)*: the lower bound because 7 is right of 4, the upper bound because 7 is in the left subtree of 6, a bound carried two levels down from the root. A parent–child check would accept this tree, since 4 < 7. The correct check visits each node once: *O(n)* time, *O(h)* stack.

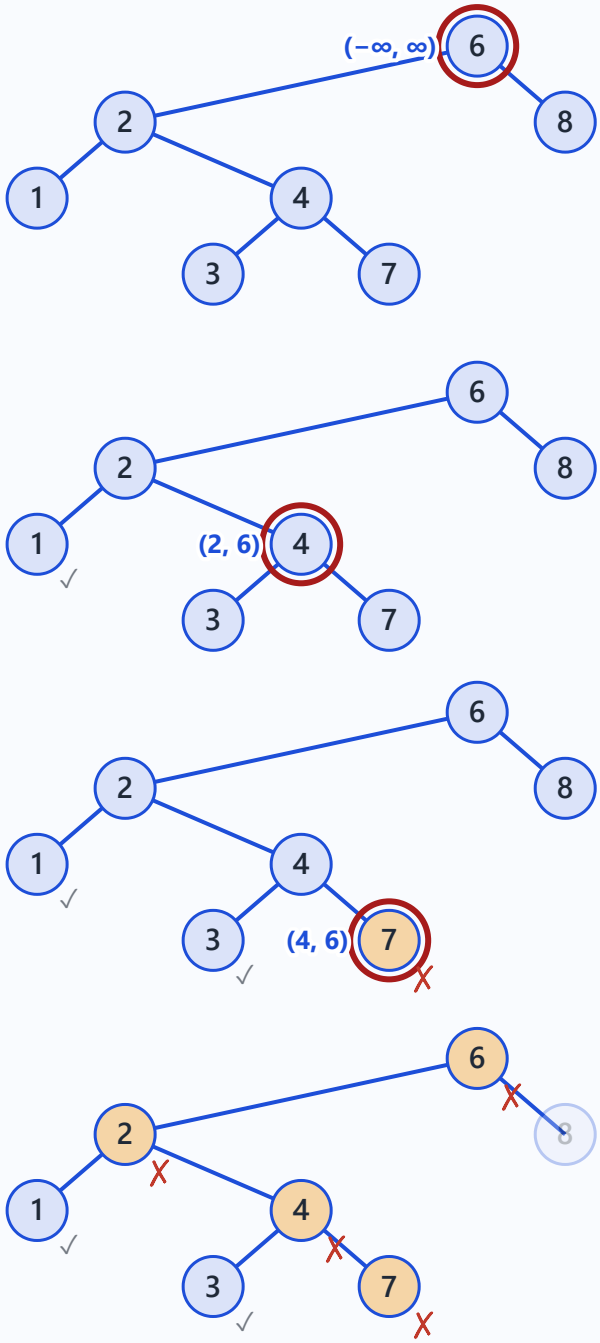


Figure 10.20: The `isBST` trace. (a) Root interval (−∞, ∞). (b) For 4, (2, 6). (c) For 7, (4, 6); 7 ≥ 6 gives `false`. (d) The tree is not a BST.

Common mistake

Starting with the `int` bounds `Integer.MIN_VALUE` and `Integer.MAX_VALUE`. If the tree holds one of those values, the strict inequality rejects a valid tree. The slides' code avoids this with `long` bounds; alternatively, pass `Integer` objects and use `null` for "no bound."

10.3.13 Application: student records

Real applications store whole records in nodes. In the slides' example the key is the student ID and the node holds the entire student object:

```
public class Student {
    int id;     // key
    String fullName;

    public Student(int id,
                   String fullName) {
        this.id = id;
        this.fullName = fullName;
    }
}
```

```
public class StudentNode {
    Student st;           // data
    StudentNode leftChild;
    StudentNode rightChild;

    public StudentNode(Student st) {
        this.st = st;
        // children start out null
    }
}
```

`Student` holds the data and `StudentNode` the tree structure, so the same student object can also live in another structure, a list for example. Insertion and search mirror the integer versions:

```
StudentNode insert(StudentNode root, Student st) {
    if (root == null) return new StudentNode(st);
    if (st.id < root.st.id)
        root.leftChild = insert(root.leftChild, st);   // continue left
    else if (st.id > root.st.id)
        root.rightChild = insert(root.rightChild, st);   // continue right
    return root;                                    // duplicate ID not inserted
}
```

```
Student search(StudentNode root, int id) {
    if (root == null) return null;                // no record
    if (id == root.st.id) return root.st; // found
    if (id < root.st.id)
        return search(root.leftChild, id);         // continue left
    return search(root.rightChild, id);             // continue right
}
```

With an `int` key, comparisons use `<`, `>` and `==` directly rather than `compareTo`. And `search` returns the student object, with the name and other fields, or `null`, instead of a `boolean`.

Example 10.4

Insert (2205, Alice Brown), (1980, Bob Davis), (2410, Carol Evans) and (2205, Duplicate) into an empty tree.

1. 2205 becomes the root.
2. 1980 < 2205: inserted to the left of the root.
3. 2410 > 2205: inserted to the right of the root.
4. 2205 is already at the root; both tests fail and the record is not inserted. "Alice Brown" stays.

`search(root, 2410).fullName` gives "Carol Evans," `search(root, 2205).fullName` "Alice Brown." `search(root, 1000)` goes left at 2205 and at 1980, hits an empty link and returns `null`.

Java's `TreeMap<Integer, Student>` does this job out of the box. It maps keys to values using a red-black tree internally: `put`, `get` and `remove` take *O(log n)*, and `higherKey`, `floorKey` and `subMap` answer successor, floor and range queries. Unlike a hand-written BST, it never degenerates, even on sorted input.

10.4 Balance and AVL trees

BST operations take *O(h)*, and *h* can reach *n − 1*. Is there a cheap way to keep the height *O(log n)* no matter what? Yes; Adelson-Velsky and Landis showed the first in 1962.

10.4.1 Why balance matters

A plain BST's shape follows the insertion order: 1, 2, 3, 4 gives a chain of height 3, yet the same keys fit in trees of height 2, such as 2(1, 3(—, 4)) and 3(1(—, 2), 4) (Figure 10.21). All three trees have inorder 1 2 3 4; they hold the same sorted set in different shapes.

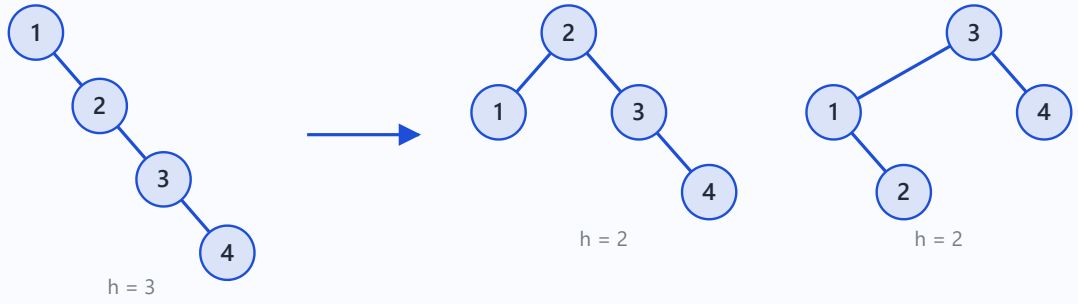


Figure 10.21: The chain 1, 2, 3, 4 (height 3) reshaped into two trees of height 2, rooted at 2 and at 3.

Definition 10.4

A **self-balancing search tree**, or simply **balanced tree**, is a BST that guarantees *O(log n)* height by repairing itself after every insertion and deletion with small local changes: rotations and recoloring.

"Local" is the key word. Rebuilding an unbalanced tree from its inorder sequence would work but costs *O(n)* per update, destroying the tree's advantage. Balanced trees touch only a few nodes on the updated path, which has length *O(log n)*, with constant work per node, so updates stay *O(log n)*.

10.4.2 A balancing example

A larger example shows local repair at work. The tree on the left of Figure 10.22 holds 11 keys; its branches 9 → 14 → 12 and 76 → 54 → 72 → 67 have grown long, giving height 4. Rearranging those two subtrees fits the keys into height 3, with 12 and 72 as new subtree roots.

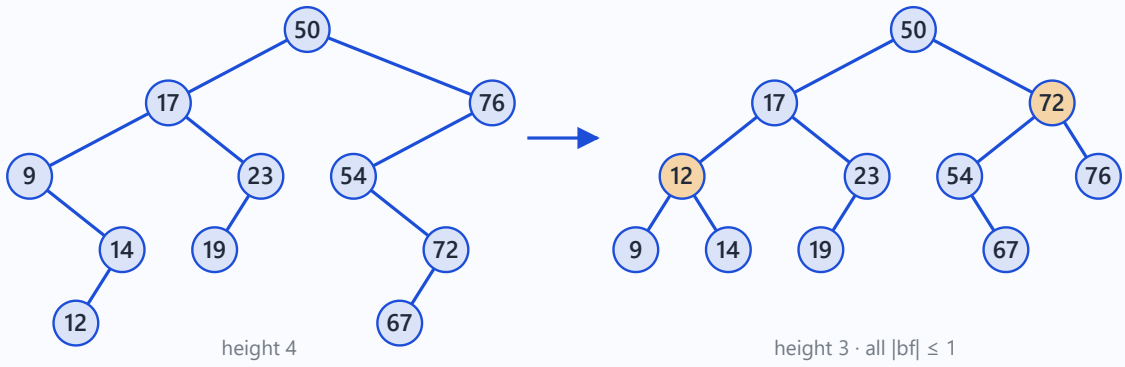


Figure 10.22: Left: an unbalanced tree of height 4. Right: the same keys at height 3, with $|bf| \le 1$ everywhere. The orange nodes, 12 and 72, are the new subtree roots.

Rearranging leaves the inorder sequence, 9 12 14 17 19 23 50 54 67 72 76, unchanged; only parent–child relations change. In the left subtree of 17, the median of 9, 14, 12 became the root with 9 and 14 as children. On the right, 54, 72, 67 and 76 were rearranged and the middle key, 72, moved up. The rest of the tree was untouched.

AVL and red-black trees make such repairs automatically. An update can disturb only its own path, and the repair runs along that path from the bottom up.

10.4.3 AVL trees and the balance factor

The AVL tree, named after the Soviet mathematicians Georgy Adelson-Velsky and Evgenii Landis and published in 1962, was the first self-balancing binary search tree. Its idea is simple: at every node, the two subtree heights differ by at most 1.

Definition 10.5

The **balance factor** of node x is $bf(x) = h(left) - h(right)$, the difference of its subtree heights, where an empty subtree has height −1. An **AVL tree** is a binary search tree with $bf \in \{-1, 0, +1\}$ at every node.

A positive balance factor means "left-heavy," a negative one "right-heavy." A leaf has $bf = -1 - (-1) = 0$; a node with only a left child, $bf = 0 - (-1) = 1$. This is why an empty subtree must count as −1: with 0, a leaf and a one-child node would look alike.

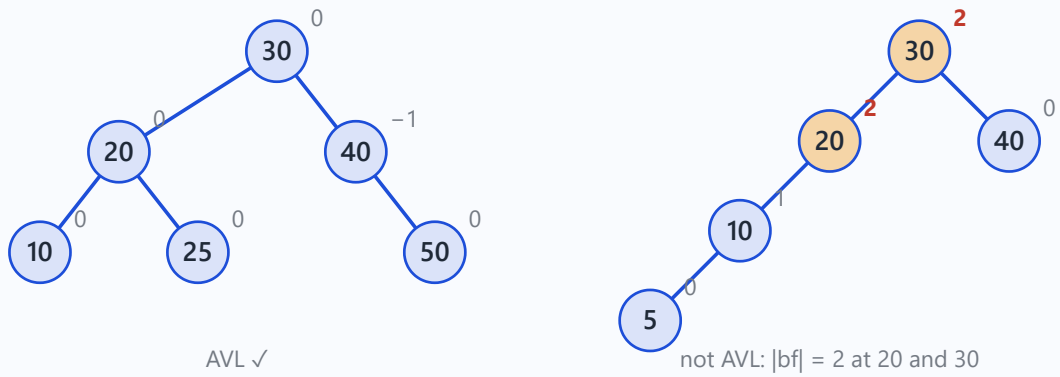


Figure 10.23: The number at each node's upper right is its balance factor. Left: a valid AVL tree, 30(20(10, 25), 40(—, 50)), with −1 at 40 and 0 elsewhere. Right: the chain 5, 10, 20 gives 20 and 30 balance factor 2; not AVL.

The right-hand tree has two unbalanced nodes. We fix the **lowest**, the one nearest the new node, here 20. That shortens its subtree by one, and 30 rebalances by itself.

Each AVL node stores its height, and balance factors are computed from the children's heights. Insertion and deletion proceed as in a plain BST; then, as the recursion unwinds, each node on the path updates its height and checks its balance factor, and a node with $|bf| = 2$ is fixed by rotation.

Common mistake

Computing the balance factor from subtree *node counts*. The AVL condition is about height. In 4(2(1, 3), 5) the root has three nodes on the left and one on the right, but heights 1 and 0, so $bf = 1$: a valid AVL tree.

10.4.4 Why is AVL height O(log n)?

The AVL condition is local, so to see that it bounds the height, ask the reverse: how *few* nodes, $N(h)$, can an AVL tree of height h have? In such a sparsest tree one subtree of the root has height $h - 1$, the other the least allowed, $h - 2$, and both are sparsest trees themselves. Hence:

$$N(0) = 1, \quad N(1) = 2, \quad N(h) = 1 + N(h - 1) + N(h - 2)$$

h	Computation	$N(h)$
0	single node	1
1	root + one child	2
2	1 + N(1) + N(0) = 1 + 2 + 1	4
3	1 + N(2) + N(1) = 1 + 4 + 2	7
4	1 + N(3) + N(2) = 1 + 7 + 4	12
5	1 + N(4) + N(3) = 1 + 12 + 7	20
6	1 + N(5) + N(4) = 1 + 20 + 12	33

Table 10.13: Minimum number of nodes in an AVL tree of height h.

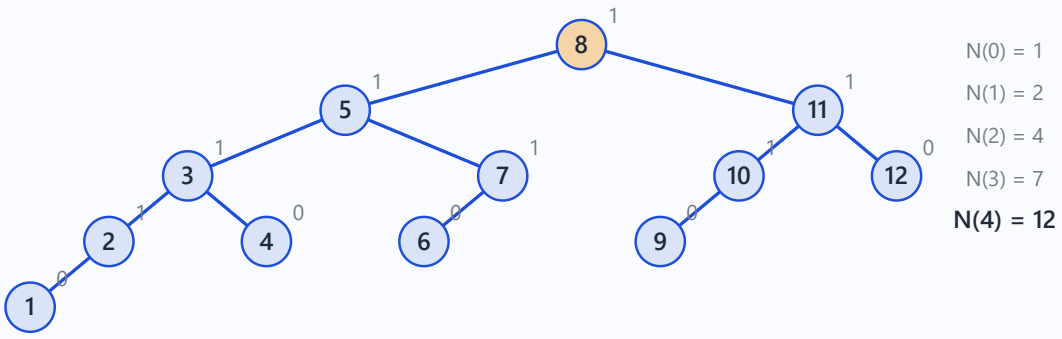


Figure 10.24: The sparsest AVL tree of height 4, with 12 nodes. The root (orange) has the sparsest trees of height 3 (7 nodes) and height 2 (4 nodes) as its subtrees. At every internal node the subtree heights differ by exactly 1.

These numbers grow like Fibonacci numbers; in fact $N(h) = F(h + 3) - 1$ (with $F(1) = F(2) = 1$), so $N(h)$ grows exponentially. A simple bound: since $N(h - 1) > N(h - 2)$, $N(h) = 1 + N(h - 1) + N(h - 2) > 2 \cdot N(h - 2)$, so the count at least doubles every two levels, and $N(h) > 2^{h/2}$.

Theorem 10.2 (AVL height)

An AVL tree with n nodes has height $h < 2 \log_2 n$; a tighter Fibonacci argument gives $h < 1.44 \log_2(n + 2)$. Search, insertion and deletion therefore take $O(\log n)$.

Proof. A tree of height h has $n \geq N(h) > 2^{h/2}$ nodes; taking base-2 logarithms, $\log_2 n > h/2$, so $h < 2 \log_2 n$. □

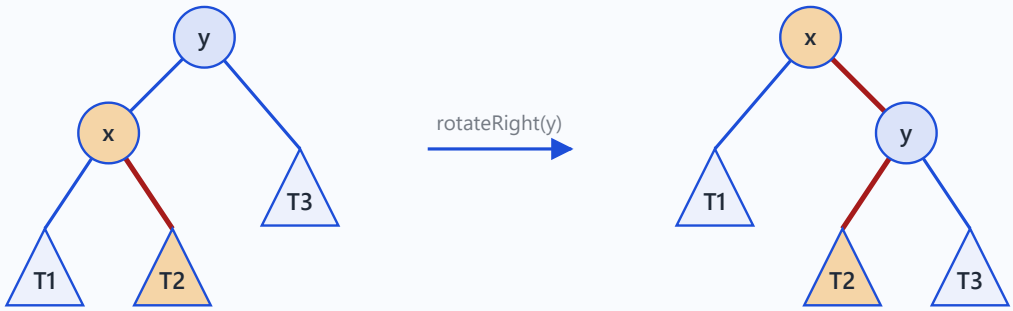
Example 10.5

How tall can a 30-node AVL tree be? Height 6 needs at least $N(6) = 33$ nodes, too many, while height 5 needs only $N(5) = 20$. The answer is 5. A plain 30-node BST can reach height 29; a perfectly balanced one has height at least 4 (the perfect 31-node tree has height 4).

So an AVL tree is at worst about 44% taller than a perfectly balanced one; with a million keys its height never exceeds 28, compared with a million-long chain for a plain BST.

10.4.5 Rotations

The basic repair tool is the **rotation**, which reverses the parent–child relation between a node and one of its children: the child moves up, the old parent down. To preserve the BST property, one subtree changes owner.



same inorder sequence on both sides: T1 · x · T2 · y · T3

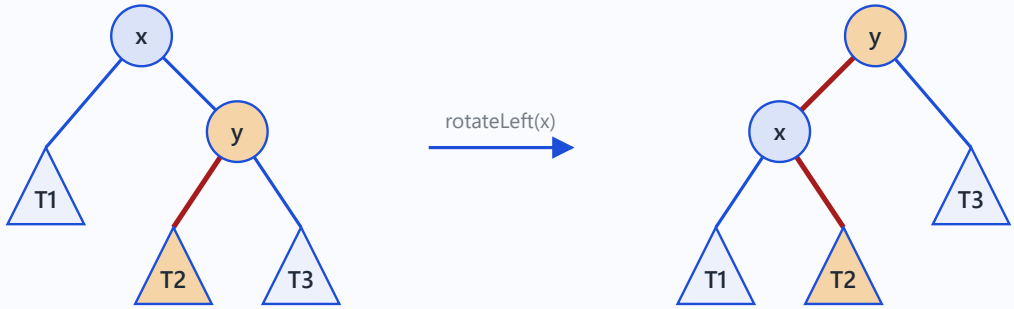


Figure 10.25: Top: right rotation at the left-heavy *y*; *x* moves up, *y* becomes its right child, and T2 moves from *x* to the left of *y*. Bottom: the mirror-image left rotation. Thick lines are changed links; the inorder sequence is T1 · x · T2 · y · T3 on both sides.

In a right rotation, *y* is the parent and *x* its left child, with subtrees T1 and T2; T3 is the right subtree of *y*. The inorder sequence is T1, *x*, T2, *y*, T3. Afterward *x* is on top with right child *y*; T1 and T3 keep their places, and T2, whose keys lie between *x* and *y*, becomes the left subtree of *y*. The sequence is unchanged.

Theorem 10.3 (Rotation)

A rotation preserves the inorder sequence and hence the BST property. It changes only three links and takes *O(1)* time; one side of the rotated subtree gets a level shorter, the other a level longer.

The three links: the left link of *y* now points to T2, the right link of *x* to *y*, and the parent's link to *x* instead of *y*. Three assignments suffice for any tree size. T1 rises a level, T3 drops one, T2 stays put, so if T1 is the heavy side of a left-heavy node, a right rotation evens things out.

Common mistake

Losing T2. If *x.right = y* comes first, the only reference to the old right subtree of *x* is gone. Move T2 to safety with `y.left = x.right` first, then write `x.right = y`.

10.4.6 Single rotations: LL and RR

Let *z* be the lowest unbalanced node after an insertion. There are four cases, named for the first two steps from *z* toward the new node (L for left, R for right). In LL the new node went into the left subtree of the left child of *z*; RR is the mirror image. Here *z* and its child lean the same way, and one rotation suffices.

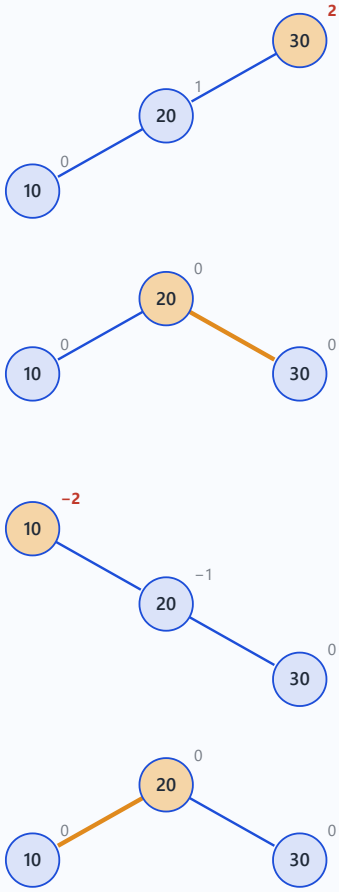


Figure 10.26: Single rotations; balance factors at upper right. (a) LL: 10 inserted left of the left child 20; *bf(30) = 2*, *bf(20) = 1*. (b) Right rotation at 30: 20 moves up, 30 becomes its right child, all balance factors 0. (c) RR: 30 inserted right of the right child 20; *bf(10) = −2*, *bf(20) = −1*. (d) Left rotation at 10.

LL shows up as *bf(z) = 2* (left-heavy) with the left child at 1 or 0; the fix is a right rotation at *z*. RR shows up as *bf(z) = −2* with the right child at −1 or 0; the fix is a left rotation at *z*.

After an insertion, a single rotation returns the subtree to its pre-insertion height. In Figure 10.26, the subtree of 30 had height 1, the insertion of 10 made it 2, and the rotation restored 1. Ancestors never see a change, so one repair per insertion suffices.

10.4.7 Double rotations: LR and RL

In the other two cases the new node ends a “zigzag.” In LR it went into the *right* subtree of the left child of *z*: *z* leans left, its child right, and the three nodes form a triangle. A single right rotation at *z* merely flips the triangle, leaving the tree unbalanced.

The fix takes two rotations: left at the child, straightening the triangle into the LL case, then right at *z*. The median of the three keys ends on top with the smaller on its left and the larger on its right. RL is the mirror image.

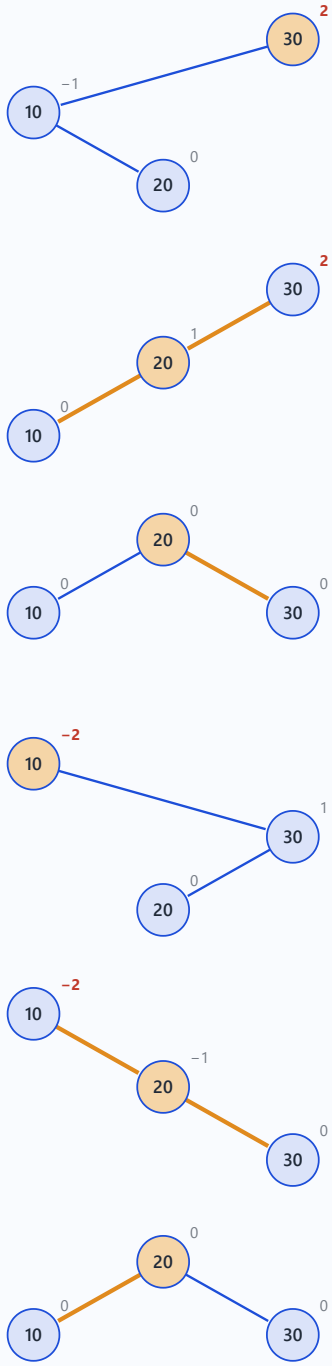


Figure 10.27: Double rotations. LR: (a) 20 inserted right of 10; *bf(30) = 2*, *bf(10) = −1*, a triangle. (b) Left at 10 gives a line (LL). (c) Right at 30: the median 20 is the root. RL: (d) 20 inserted left of 30; *bf(10) = −2*, *bf(30) = 1*. (e) Right at 30 gives a line (RR). (f) Left at 10: 20 is the root.

Case	*bf(z)*	Child's *bf*	Fix
LL	+2	left child ≥ 0	rotate right at *z*
RR	−2	right child ≤ 0	rotate left at *z*

Case	*bf(z)*	Child's *bf*	Fix
LR	+2	left child < 0	left at the left child, then right at *z*
RL	−2	right child > 0	right at the right child, then left at *z*

Table 10.14: The four imbalance cases in an AVL tree (*z*: the lowest unbalanced node).

> **Common mistake**
>
> Choosing the case from *bf(z)* alone. *bf(z) = 2* only says "left-heavy"; LL versus LR depends on the child. Inserting 50, 20, 30 gives *bf(50) = 2* and *bf(20) = −1*: LR. A single right rotation at 50 yields 20(—, 50(30, —)) with *bf(20) = −2*, still unbalanced; the right answer is 30(20, 50).

10.4.8 The AVL code

An AVL node holds a key, a height and two child references. The slides first give three helpers and the right rotation:

```
int h(AvlNode d) {
    return d == null ? -1 : d.height;
}
int bf(AvlNode d) {
    return h(d.left) - h(d.right);
}
void update(AvlNode d) {
    d.height = 1 +
        Math.max(h(d.left), h(d.right));
}
AvlNode rotateRight(AvlNode y) {
    AvlNode x = y.left;
    y.left = x.right;  // T2 → y.left
    x.right = y;       // y → x.right
    update(y);         // lower node first
    update(x);
    return x;          // new root
}
```

`h` returns −1 for an empty subtree, sparing other methods a `null` check; `bf` is the difference of the children's heights; `update` recomputes a node's height from its children's, which must already be correct, so heights are always updated bottom-up.

`rotateRight(y)` codes Figure 10.25: `x` is the rising left child, `y.left = x.right` moves T2, and `x.right = y` reverses the relation. `y` is updated before `x`, since `y` is now the child and the height of `x` depends on it. The method returns the new subtree root `x`, which the caller assigns to the parent's link, the third link.

The slides abbreviate the left rotation as the "left ↔ right mirror"; in full:

```
AvlNode rotateLeft(AvlNode x) {
    AvlNode y = x.right;
    x.right = y.left;  // T2 → x.right
    y.left = x;        // x → y.left
    update(x);         // lower node first
    update(y);
    return y;          // new root
}
```

`rebalance` combines the four cases:

```
AvlNode rebalance(AvlNode d) {
    update(d);
    if (bf(d) > 1) {       // left-heavy
        if (bf(d.left) < 0)  // LR
            d.left = rotateLeft(d.left);
        return rotateRight(d); // LL
    }
    if (bf(d) < -1) {      // right-heavy
        if (bf(d.right) > 0)  // RL
            d.right = rotateRight(d.right);
        return rotateLeft(d); // RR
    }
    return d;
}
// rotateLeft: mirror of left ↔ right
// insert/delete as in a BST; on return:
//   return rebalance(root);
```

`rebalance(d)` updates the height of `d`. If the node is left-heavy ($bf > 1$) and its left child leans right, the case is LR: rotate left at the child, storing the result in `d.left`. In both cases, then rotate right at `d`. The right-heavy case mirrors this; a balanced node is returned as is.

Insertion changes only the last line of the BST version:

```
AvlNode insert(AvlNode d, int k) {
    if (d == null) return new AvlNode(k);
    if (k < d.key)
        d.left = insert(d.left, k);
    else if (k > d.key)
        d.right = insert(d.right, k);
    else return d;          // no duplicates
    return rebalance(d);
}
```

`return rebalance(d)` replaces `return d`, so as the recursion unwinds, every node from the new leaf to the root updates its height and is fixed on the spot if unbalanced. Deletion likewise adds `rebalance` to the returns of BST deletion.

The descent covers *O(log n)* levels with constant work per `rebalance` and rotation, so insertion and deletion take *O(log n)*. Insertion needs at most one repair (a single or double rotation). In deletion a repair may shorten the subtree and push the imbalance up, so there may be a rotation at every level, *O(log n)* in all.

10.4.9 Code trace: rotateRight and rotateLeft

First an LL case: inserting 5 into 40(20(10, 30), 50). 5 goes down 40 → 20 → 10 and lands left of 10. On the way back `rebalance(10)` and `rebalance(20)` find *bf* 1 and do nothing; `rebalance(40)` finds *bf = 2* with left child 20 at 1, so it calls `rotateRight(40)`.

Line	Effect	Changed
`x = y.left`	the child that moves up is 20; no link has changed yet	—
`y.left = x.right`	the left link of 40 now points to T2, that is, 30; only `x` still reaches 20	link 1
`x.right = y`	the right link of 20 points to 40 instead of 30	link 2
`update(y)`	40.height = 1 + max(0, 0) = 1 (was 3)	height
`update(x)`	20.height = 1 + max(1, 1) = 2 (unchanged, but recomputed)	height
`return x`	the new subtree root is 20; the caller assigns it to its own link	link 3

Table 10.15: Trace of `rotateRight(y)` with `y` = 40 (LL case).

The result is 20(10(5, —), 40(30, 50)), with balance factors 0 at 20 and 40, 1 at 10, 0 at the leaves. The inorder sequence 5 10 20 30 40 50 is unchanged. 40 dropped from height 3 to 1, and the new subtree root 20 has height 2, as the subtree did before the insertion.

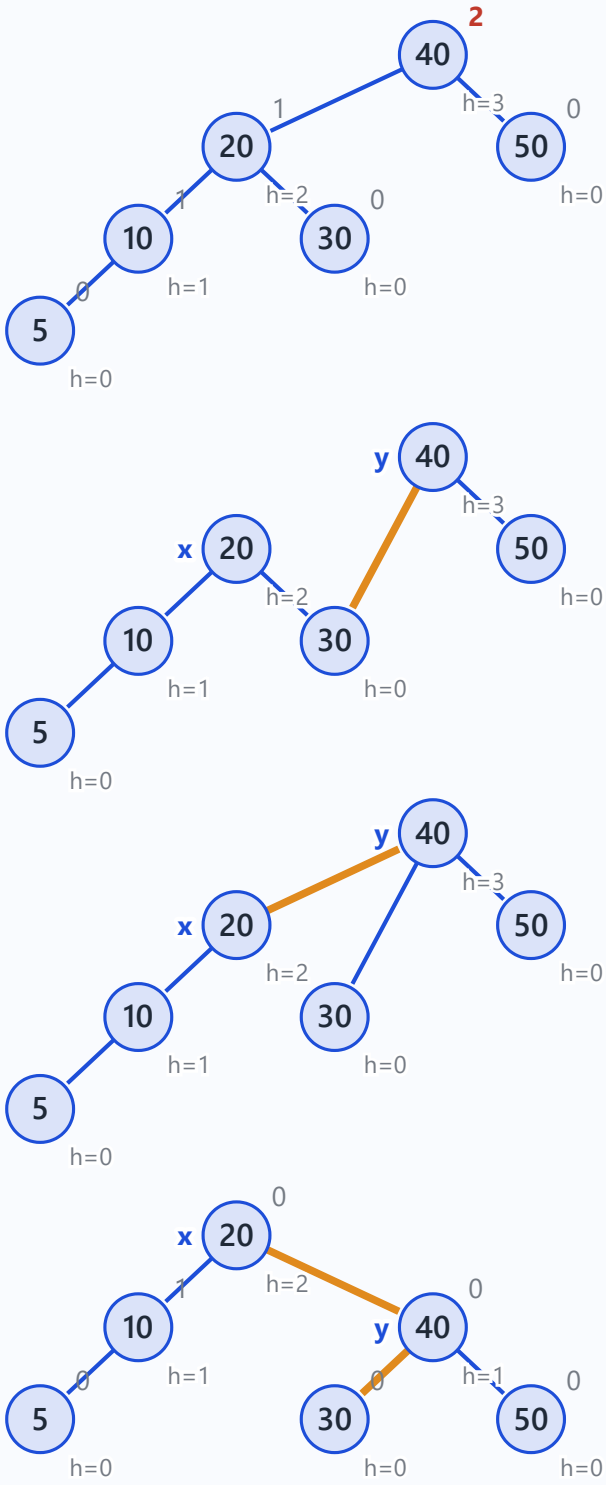


Figure 10.28: `rotateRight(40)`, with balance factor and height beside each node. (a) After inserting 5: *bf(40) = 2*, *bf(20) = 1*, LL. (b) `y.left = x.right` : 40's left link points to 30. (c) `x.right = y` : 40 is the right child of 20. (d) New subtree root 20; all balance factors in {−1, 0, 1}.

The mirror-image RR case: inserting 60 into 20(10, 40(30, 50)). `rebalance(20)` finds *bf* = −2 with right child 40 at −1 and calls `rotateLeft(20)`: `y = x.right` is 40, `x.right = y.left` points 20 at 30, `y.left = x` makes 20 the left child of 40, and the updates give 20 height 1 + max(0, 0) = 1 (was 3) and 40 height 1 + max(1, 1) = 2. The result is 40(20(10, 30), 50(—, 60)).

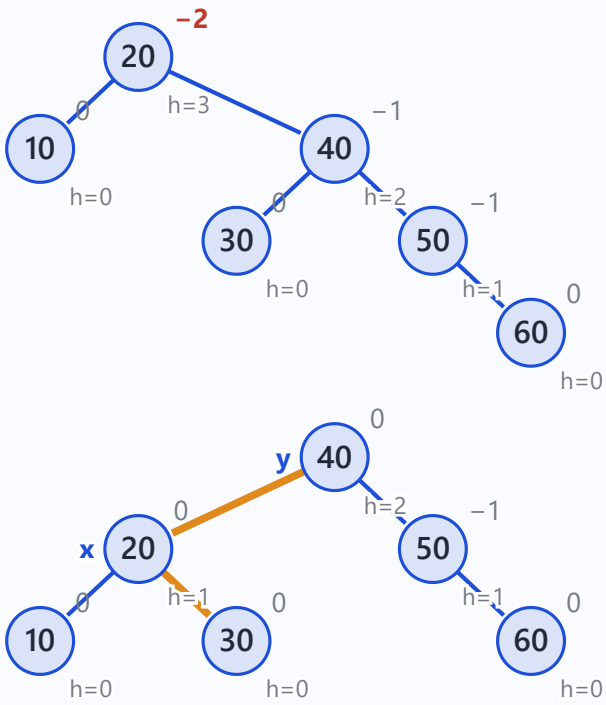


Figure 10.29: `rotateLeft(20)`. (a) After inserting 60: *bf(20) = −2*, *bf(40) = −1*, RR. (b) New subtree root 40, with 20 as its left child and 30 as the right child of 20.

Common mistake

Updating heights in the wrong order after a rotation. Calling `update(x)` first computes the height of `x` from the stale height of `y` (3 in the LL example), giving 4; the balance factors above are then wrong and trigger needless rotations. Update the lowered node first, then the new root.

10.4.10 Code trace: rebalance

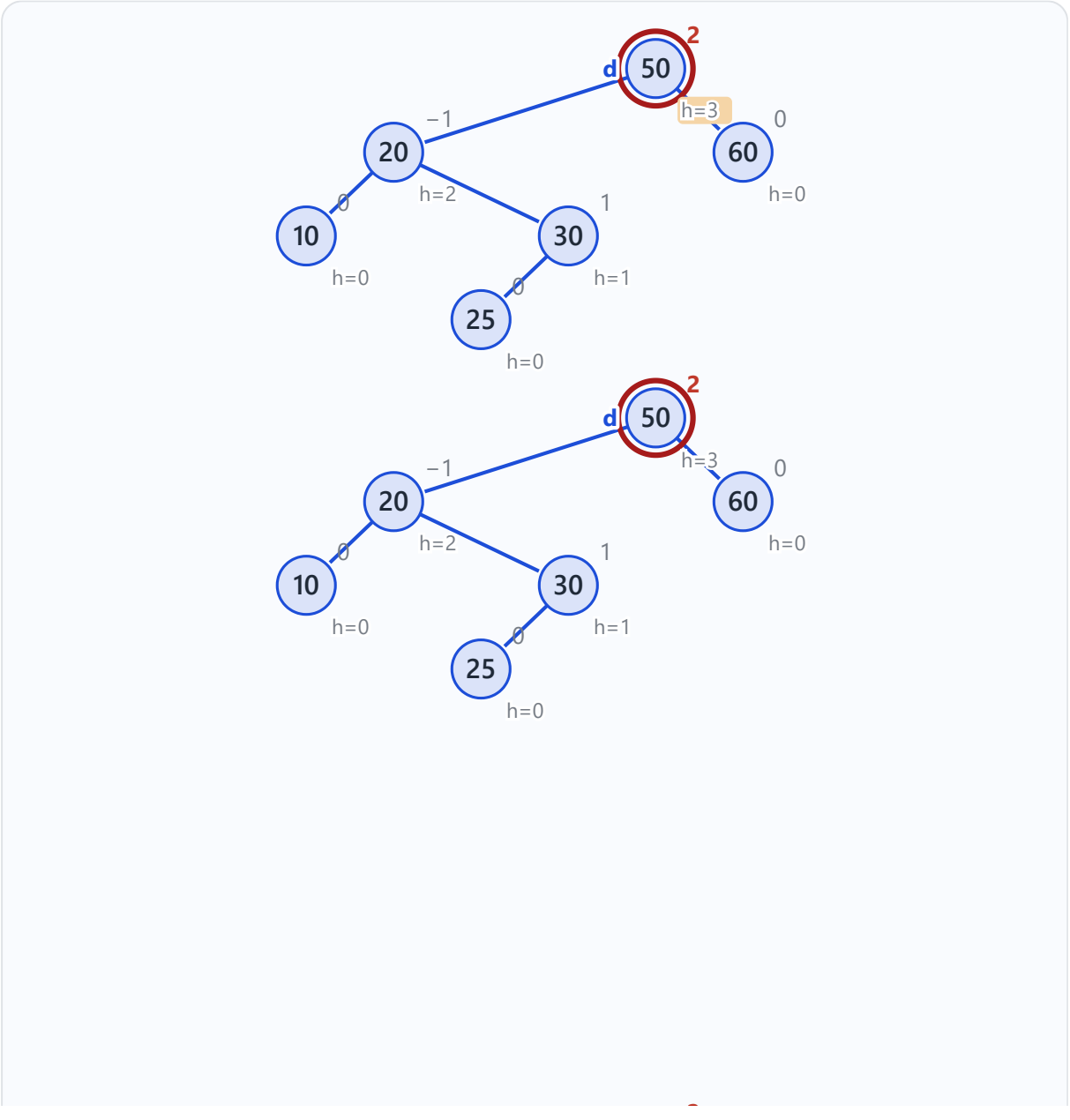
Now a double rotation in code, the LR case: inserting 25 into 50(20(10, 30), 60). 25 goes down 50 → 20 → 30 and lands left of 30, and `rebalance` runs at each node on the way back.

Call	Height	*bf*	Decision
`rebalance(30)`	0 → 1	1	no rotation, `return d`

Call	Height	*bf*	Decision
`rebalance(20)`	1 → 2	0 − 1 = −1	no rotation, `return d`
`rebalance(50)`	2 → 3	h(20) − h(60) = 2 − 0 = 2	left-heavy; *bf(20) = −1 < 0* → LR
`d.left = rotateLeft(20)`	30 rises; 25 moves right of 20		triangle → line; *bf(50)* still 2
`return rotateRight(50)`	30 becomes the root; 50 goes to its right and takes its old right (`null`) as left		new subtree root 30

Table 10.16: The `rebalance` calls on the way back after inserting 25 (LR case).

The result is 30(20(10, 25), 50(—, 60)), back to the pre-insertion height 2, with balance factors 0, 0, −1 at 30, 20, 50. Assigning the first rotation's result to `d.left` is essential: only then does 50 point to the new subtree root 30.



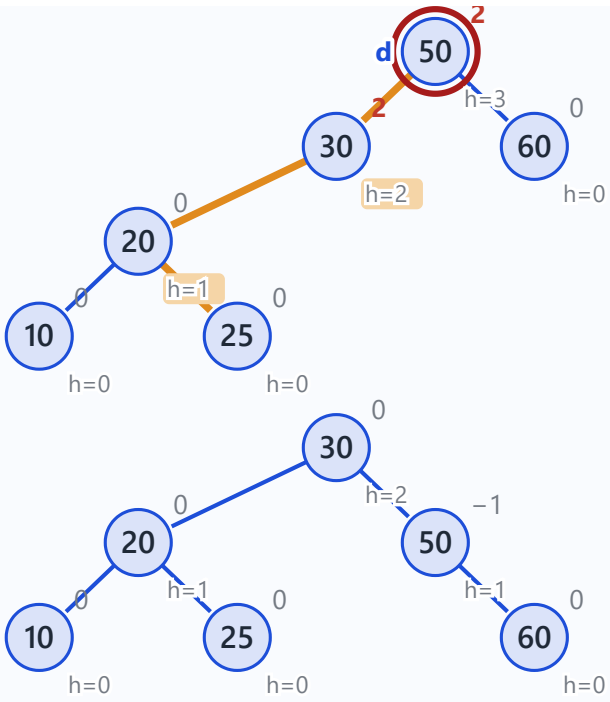


Figure 10.30: `rebalance(50)`. (a) Height 3, *bf = 2*. (b) *bf(20) = −1*: the left child leans right, LR. (c) `rotateLeft(20)`: 30 moves up, the triangle becomes a line. (d) `rotateRight(50)`: 30 is the new subtree root; height 3 → 2.

10.4.11 An AVL insertion example

Insert 10, 20, 30, 40, 50, 25 into an empty AVL tree, input on which a plain BST would build (except for the last key) a right-leaning chain. Table 10.17 shows each step.

Insert	Descent	Imbalance on the way back	Rotation	Resulting tree
10	empty tree	—	—	10
20	10 → right	*bf(10) = −1*, fine	—	10(—, 20)
30	10 → 20 → right	*bf(10) = −2*, *bf(20) = −1*: RR	left at 10	20(10, 30)
40	20 → 30 → right	*bf(30) = −1*, *bf(20) = −1*, fine	—	20(10, 30(—, 40))
50	20 → 30 → 40 → right	*bf(30) = −2*, *bf(40) = −1*: RR	left at 30	20(10, 40(30, 50))
25	20 → 40 → 30 → left	*bf(20) = −2*, *bf(40) = 1*: RL	right at 40, left at 20	30(20(10, 25), 40(—, 50))

Table 10.17: Inserting 10, 20, 30, 40, 50, 25 into an empty AVL tree.

Can 30 and 20 both be unbalanced when 50 arrives? No. `rebalance(40)` runs first, then `rebalance(30)`, which fixes 30 and restores its subtree's old height of 1. By the time `rebalance(20)` runs, its height stays 2 and *bf(20) = −1*: fixing the lowest unbalanced ancestor leaves everything above untouched.

The last key, 25, lands left of 30. *bf(30) = 1* and *bf(40) = 1* are fine, but *bf(20) = h(10) − h(40) = 0 − 2 = −2* with the right child 40 leaning left: RL. Rotating right at 40 (30 moves up) and then left at 20 brings 30 to the root and moves 25 right of 20. Six insertions give height 2; a plain BST would have height 4.

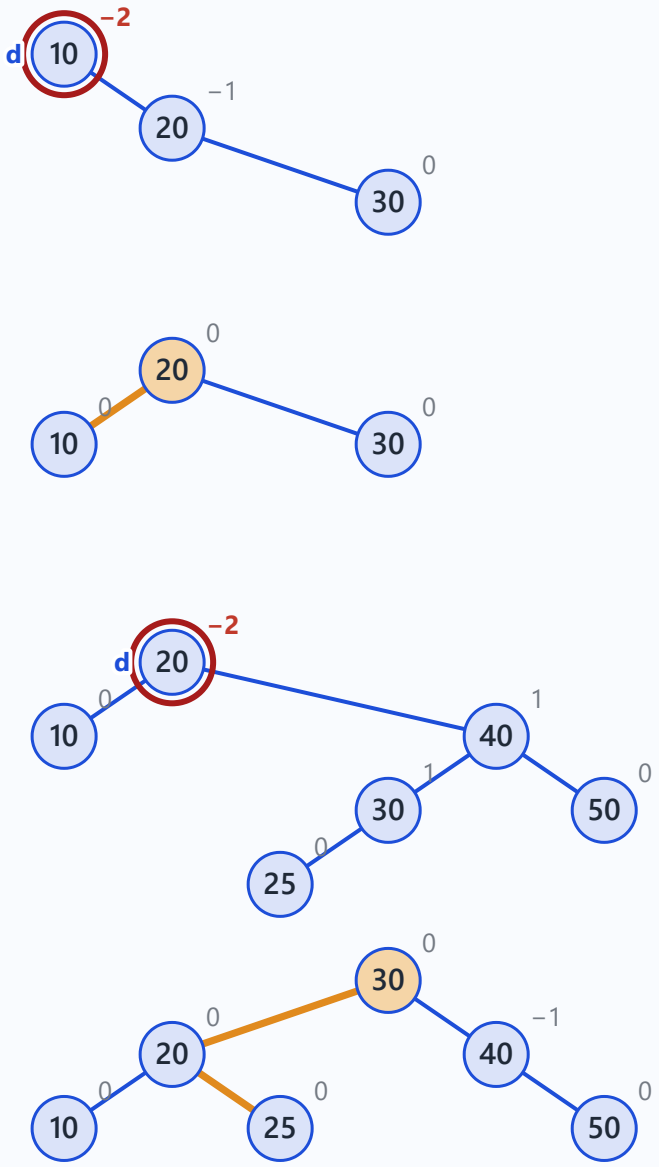


Figure 10.31: Four moments of the AVL trace. (a) 30 inserted: *bf(10) = −2*, RR. (b) Left rotation at 10; 20 at the root. (c) 25 inserted: *bf(20) = −2*, *bf(40) = 1*, RL. (d) After two rotations 30 is the root; all balance factors in {−1, 0, 1}.

Example 10.6

What happens when 1, 2, ..., 7 are inserted in order into an empty AVL tree? Every rotation is RR.

1. Inserting 3 triggers a left rotation at 1: 2(1, 3).
2. Insert 4: 2(1, 3(—, 4)); no imbalance.
3. Inserting 5 triggers a left rotation at 3: 2(1, 4(3, 5)).
4. Inserting 6 makes $bf = -2$ at the root 2; a left rotation at 2 gives 4(2(1, 3), 5(—, 6)).
5. Inserting 7 triggers a left rotation at 5: 4(2(1, 3), 6(5, 7)).

Four rotations produce the perfect tree of height 2. Sorted input, a plain BST's worst case, gives the AVL tree its best shape.

10.5 Red-black trees

An AVL tree keeps a tight balance, and pays for it: after a deletion the imbalance can climb to the root, with a rotation at every level. A red-black tree accepts looser balance in exchange for very few rotations, doing most repairs by recoloring. Guibas and Sedgewick defined it in today's color terms in 1978; Java's `TreeMap` and `TreeSet` and C++'s `std::map` use it.

10.5.1 The rules

Definition 10.6

A **red-black tree** is a binary search tree whose nodes are colored red or black, subject to five rules:

1. Every node is red or black.
2. The root is black.
3. All leaves, the empty children (NIL), count as black.
4. A red node has black children: no two reds in a row.
5. Every path from a node down to the NIL leaves below it has the same number of black nodes, the node's **black height**.

Rule 3 just means that in code `null` children count as black, so rules 4 and 5 still make sense where a child is missing. When counting black nodes we count the root but not the NIL leaves.

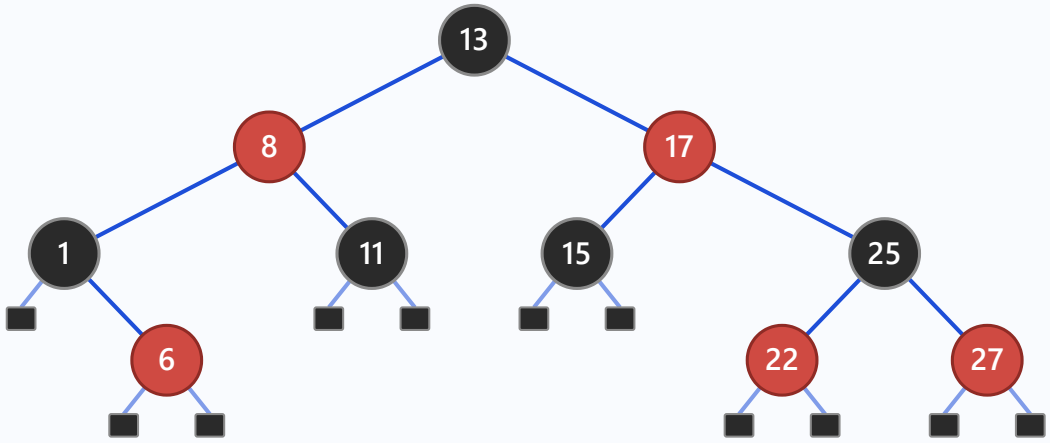


Figure 10.32: A valid red-black tree; small black squares are NIL leaves. Black: 13 (root), 1, 11, 15, 25. Red: 8, 17, 6, 22, 27. Every root-to-NIL path has 2 black nodes (root included, NIL excluded).

Check the rules on Figure 10.32. The red 8 and 17 have black children (1, 11 and 15, 25), and the red leaves 6, 22, 27 have NIL children, so rule 4 holds. For rule 5: 13 → 8 → 1 → NIL has blacks 13 and 1; 13 → 8 → 1 → 6 → NIL the same, since 6 is red; 13 → 17 → 25 → 27 → NIL has 13 and 25. Two on every path.

Common mistake

Checking rule 5 only on the longest and shortest paths. Black counts must match on *every* path down to NIL, including the empty side of a one-child node. Take a black root 20 with black children 10 and 30, and a black 25 as left child of 30: 20 → 10 → NIL has 2 blacks, 20 → 30 → 25 → NIL has 3. Rule 5 fails.

10.5.2 How the rules guarantee balance

The height bound comes from two rules acting together. Let b be the root's black height. Every root-to-NIL path has exactly b black nodes, so the shortest, all black, has b nodes. The longest also has b blacks, with reds in between, but rule 4 allows at most one red after each black, so it has at most $2b$ nodes.

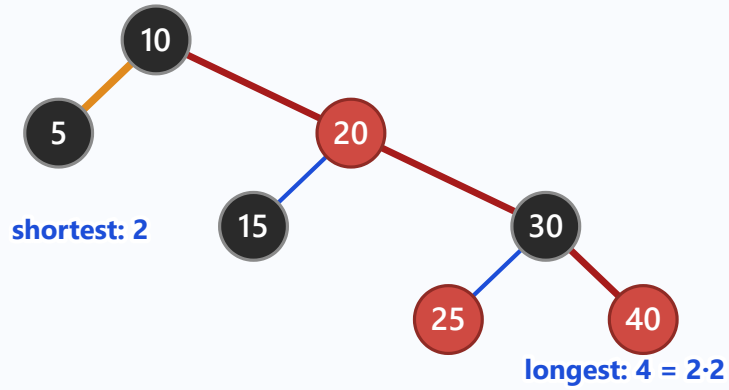


Figure 10.33: With black height 2, the shortest path is 10 → 5 (2 nodes, both black) and the longest 10 → 20 → 30 → 40 (4 nodes, alternating colors): twice as long.

Theorem 10.4 (Height of a red-black tree)

In a red-black tree the longest root-to-NIL path is at most twice the shortest, and with n internal nodes the height is $h \le 2\log_2(n+1)$.

Proof. The first claim follows from the count above. For the second, a node of black height b has at least $2^b - 1$ internal nodes in its subtree, by induction: black height 0 means a NIL, with $2^0 - 1 = 0$ nodes; otherwise each child has black height at least $b - 1$ and thus at least $2^{b-1} - 1$ nodes, for a total of at least $2(2^{b-1} - 1) + 1 = 2^b - 1$. At the root rule 4 gives $b \ge h/2$, so $n \ge 2^{h/2} - 1$ and $h \le 2\log_2(n+1)$. □

Search, insertion and deletion thus take $O(\log n)$. Fix-up uses at most 2 rotations after an insertion and 3 after a deletion; the rest is recoloring. The height bound is looser than AVL's (about $1.44\log_2 n$), so the tree can be somewhat taller, but updates are cheaper and each node needs only one extra bit.

10.5.3 Insertion and the fix-up cases

Insertion has two phases. First the new node is attached as a leaf by BST insertion and colored **red**: a black node would add a black to its paths and break rule 5, whereas a red one changes no black height. Only rule 4 can break, if the parent is red too. The second phase repairs that.

The fix-up involves four nodes (Figure 10.34): the new node x, its parent p, grandparent g and **uncle** u, the parent's sibling. A red parent cannot be the root, so g exists, and it is black, or two reds would already have been adjacent. The uncle's color decides what happens.

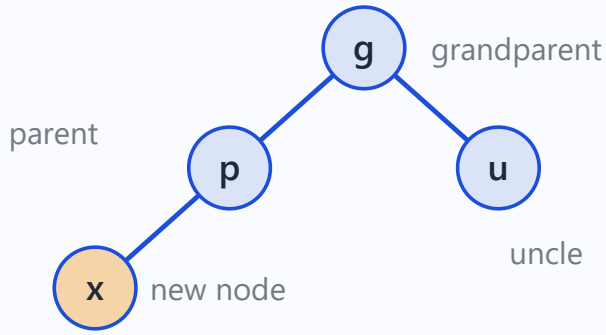


Figure 10.34: The roles in the fix-up: new node *x*, parent *p*, grandparent *g*, uncle *u*.

Case	Action	Then
Red uncle	Recolor: *p* and *u* black, *g* red.	The problem may move up to the grandparent; set *x* = *g* and continue the loop.
Black uncle, triangle	If *x*–*p*–*g* zigzags, rotate at *p*; *x* and *p* swap roles.	Becomes the line case.
Black uncle, line	Rotate at *g*; the old parent becomes black, the old grandparent red.	Done.

Table 10.18: Fix-up cases after an insertion (when the parent is red).

With a black parent nothing needs fixing. At the end the root is always colored black, and a NIL uncle counts as black.

Recoloring preserves rule 5: *g* goes from black to red and *p*, *u* from red to black, and every path down from *g* passes through exactly one of *p*, *u*, so it still has one black here. But a red *g* may now have a red parent, breaking rule 4 higher up, so the loop continues from *g*. The problem climbs two levels per step and at worst ends at the root, which is then colored black.

Recoloring is no use when the uncle is black: making *g* red would remove a black from the uncle's paths. Rotate instead. In the line case (*x* and *p* are children on the same side), rotate at *g*; *p* rises and turns black, *g* drops and turns red. In the triangle case, first rotate at *p* to get a line, just like AVL single and double rotations. The subtree root is then black and the loop ends.

10.5.4 The fix-up code

A red-black node has, besides the key and children, a color and a parent reference that lets the fix-up climb. The slides' fix-up code:

```
void fixUp(Node x) {       // x: new, red
    while (x != root && isRed(x.parent)) {
        Node p = x.parent, g = p.parent;
        Node u = (p == g.left) ? g.right : g.left;
        if (isRed(u)) {           // red uncle
            p.color = u.color = BLACK;
            g.color = RED;
            x = g;                // problem moves to g
        } else {                  // black uncle
            if (isTriangle(x, p, g)) { // zigzag
                rotate(p, x);     // make it a line
                Node t = x; x = p; p = t;
            }
            rotate(g, p);         // line
            p.color = BLACK; g.color = RED;
        }
    }
    root.color = BLACK;
}
```

The helpers and the rest of the insertion are given as comments:

```
// Insert: add as a leaf as in a BST,
// link the parent, color it red:
//   x = bstInsert(key);
//   x.color = RED; fixUp(x);

// NIL (null) counts as black
boolean isRed(Node n) {
    return n != null
        && n.color == RED;
}
// is x-p-g a zigzag (triangle)?
boolean isTriangle(Node x, Node p,
                   Node g) {
    return (p == g.left)
        != (x == p.left);
}
// rotate(a, b): b is a child of a;
// moves b up into a's place (right
// rotation if b is on the left, left
// if on the right; parent links too)
```

`isRed(n)` treats `null` as black, so NIL uncles need no special case. `isTriangle(x, p, g)` compares "is p a left child?" with "is x a left child?"; different answers mean a zigzag. `rotate(a, b)` lifts b, a child of a, into the place of a (right rotation if b is on the left, left if on the right), updating parent links. The traces cite lines of `fixUp`: 2 is the loop test, 5 the uncle check, 10 the triangle check, 14 the rotation at g, and 18 the coloring of the root.

`Node t = x; x = p; p = t;` swaps roles after the triangle rotation, which moved *x* up and *p* down, so that `rotate(g, p)` rotates the upper node.

BST insertion is *O(log n)*, and the loop climbs at least two levels per iteration, so it runs at most *O(log n)* times. Rotations happen only in the last iteration, since the black-uncle branch ends the loop, and number at most two.

Common mistake

Forgetting `root.color = BLACK` after the loop. Recoloring can turn the root red, after which the loop stops because *x* is the root, leaving rule 2 broken. This line is also the only way the black height of the whole tree grows.

10.5.5 Code trace: 10, 20, 30, 15

Insert 10, 20, 30, 15 into an empty red-black tree. A letter after a key gives its color, R (red) or B (black): R, B.

Insert	Loop	*p*, *g*, *u*	Action	Result
10	x is the root → not entered	—	line 18: root black	10B
20	parent 10 is black → not entered	—	—	10B(—, 20R)
30	parent 20 is red → entered	20, 10, NIL (black)	line: `rotate(10, 20)` to the left; 20 black, 10 red	20B(10R, 30R)
15	parent 10 is red → entered	10, 20, 30R	red uncle: 10 and 30 black, 20 red; *x* = *20*; x is the root → loop ends; line 18: 20 black	20B(10B(—, 15R), 30B)

Table 10.19: Trace of `fixUp` while inserting 10, 20, 30, 15.

For 30 the uncle is NIL, hence black, and 10 → 20 → 30 is a line, so `isTriangle` is false and line 14 rotates left at the grandparent, as in the AVL RR fix. After recoloring, the subtree root 20 is black and the loop ends.

For 15, the uncle 30 is red, so recoloring suffices. The grandparent 20 turns red and the loop moves to it, but 20 is the root, so the loop stops and line 18 makes it black again. The tree's black height rose from 1 to 2.

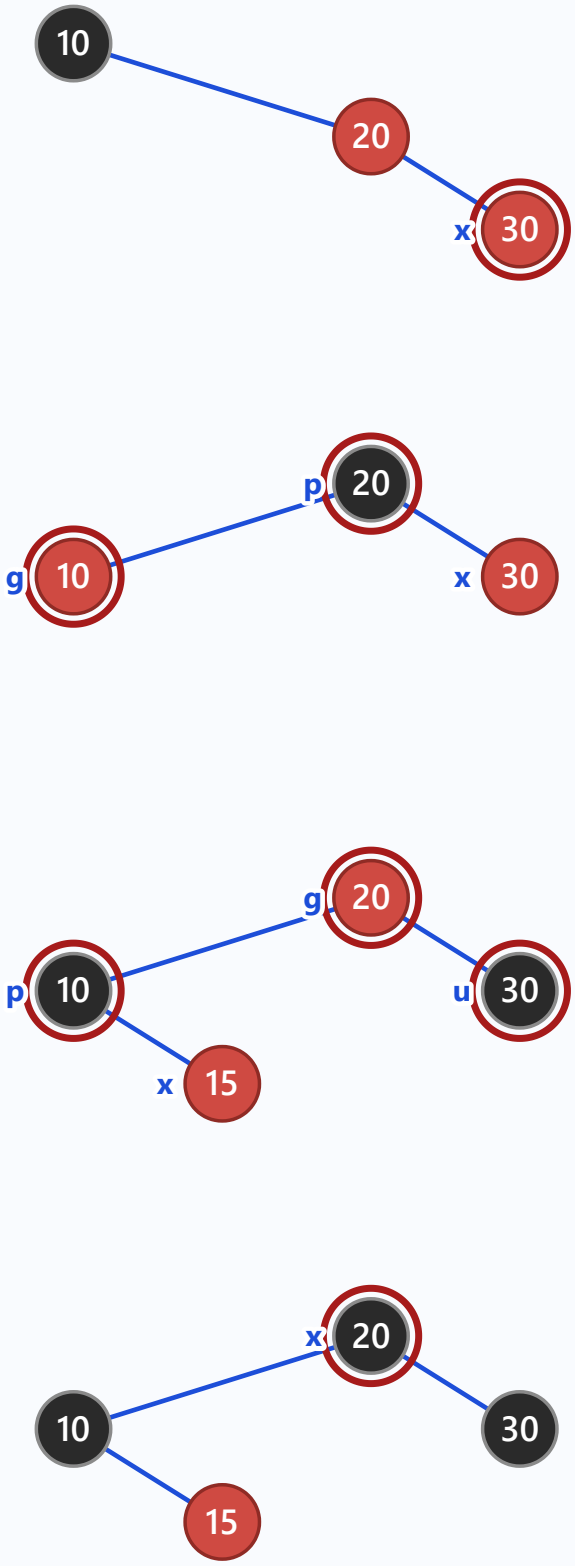


Figure 10.35: Four moments of the first trace. (a) 30 inserted red under the red 20. (b) After the left rotation at 10 and recoloring: 20 black, 10 and 30 red. (c) 15 inserted; the uncle 30 is red, so only colors change: 10, 30

black, 20 red. (d) The root is black again; black height +1.

10.5.6 Code trace: 12 and 5

Next insert 12, then 5, into 20B(10B(—, 15R), 30B): 12 shows the triangle case, 5 recoloring that moves the problem up.

Insert	Position	*p*, *g*, *u*	Action	Result
12	left child of 15	15R, 10B, NIL	black uncle, triangle: `rotate(15, 12)` to the right; roles swap (x = 15, p = 12); `rotate(10, 12)` to the left; 12 black, 10 red	20B(12B(10R, 15R), 30B)
5	left child of 10	10R, 12B, 15R	red uncle: 10 and 15 black, 12 red; *x = 12*; parent 20 is black → loop ends	20B(12R(10B(5R, —), 15B), 30B)

Table 10.20: Trace of `fixUp` while inserting 12 and 5.

For 12, 15 is the *right* child of 10 and 12 the *left* child of 15: a zigzag. Rotating right at 15 makes the line 10 → 12 → 15, though two reds remain adjacent; after the role swap, rotating left at 10 lifts 12 to the subtree root, black, with 10 red. This is the AVL RL fix.

For 5, the uncle 15 is red. Recoloring turns 12 red, and the loop stops at 12 because its parent 20 is black. Every path now has 2 black nodes: 20 and 10 on 20 → 12 → 10 → 5 → NIL, 20 and 30 on 20 → 30 → NIL.

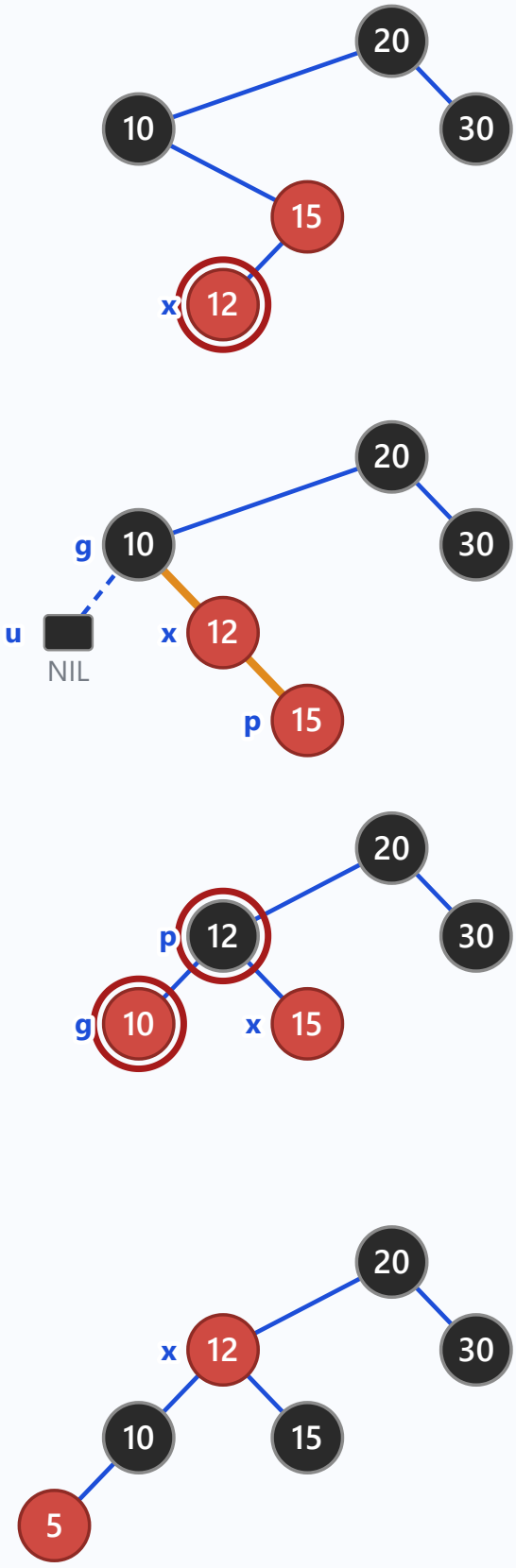


Figure 10.36: Four moments of the second trace. (a) 12 inserted as left child of the red 15. (b) Right rotation at 15: the triangle becomes a line. (c) Left rotation at 10 and recoloring: 12 black, 10 and 15 red. (d) 5

inserted and recolored; all rules hold: 20B(12R(10B(5R, —), 15B), 30B).

Example 10.7

Insert 1 to 10 in order into an empty red-black tree. The code performs five recolorings and five rotations and produces 4B(2B(1B, 3B), 6B(5B, 8R(7B, 9B(—, 10R)))), of height 4 and black height 3. A plain BST would give a chain of height 9, an AVL tree height 3: the red-black tree is one level taller than AVL, but both are logarithmic.

10.5.7 Deletion in balanced trees

Deletion in both AVL and red-black trees begins with plain BST deletion; a two-child node is replaced by its successor or predecessor, so the node actually removed is always a leaf or has one child. Its removal may break the AVL height condition or the red-black black-height rule.

- **AVL repair:** Update heights from the removed node up to the root, rotating (LL, RR, LR or RL) at every node with $|bf| = 2$. Unlike insertion, a rotation can leave the subtree shorter than before and unbalance an ancestor, so several ancestors may need repair.
- **Red-black repair:** Deleting a red node changes no black counts. Removing a black node leaves its paths one black short, which recoloring and rotations fix according to the colors of the sibling and its children. Textbooks call this the “double black” fix-up; the deficit can climb to the root, but there are at most three rotations.

Structure	Checking upward	Rotations	Total
AVL	$O(\log n)$	up to $O(\log n)$	$O(\log n)$
Red-black	$O(\log n)$	at most 3	$O(\log n)$

Table 10.21: Cost of deletion in balanced trees.

Neither insertion nor deletion disturbs the BST order. Balancing changes only the tree's shape and the auxiliary data (height or color), so an inorder traversal always gives the same sorted list.

10.5.8 AVL or red-black?

Both perform every operation in $O(\log n)$; the right choice depends on the workload. Table 10.22 summarizes the differences.

Property	AVL	Red-black
Balance criterion	$\lvert bf\rvert \le 1$ at every node	color rules, equal black height
Maximum height	$\approx 1.44 \log_2 n$	$2 \log_2(n + 1)$
Search	$O(\log n)$, slightly faster (shorter tree)	$O(\log n)$
Rotations per insertion	at most 2 (one single or one double)	at most 2

Property	AVL	Red-black
Rotations per deletion	up to *O(log n)*	at most 3
Extra data per node	height (or balance factor)	1-bit color
Preferred for	search-dominated workloads	frequent insertions and deletions, libraries

Table 10.22: AVL and red-black trees compared.

In practice the difference is often small. A dictionary built once and queried millions of times gains a few comparisons per search from AVL's shorter tree; a general-purpose library class with frequent updates benefits from red-black's cheaper ones, which is why the Java and C++ standard libraries use red-black trees.

10.6 Splay trees

AVL and red-black trees store balance information in each node and use it to repair the tree after each operation. A splay tree stores none and does not even try to stay balanced; instead, each access moves the accessed node to the root. Proposed by Sleator and Tarjan in 1985, it is also called a self-adjusting BST.

10.6.1 A self-adjusting tree

> **Definition 10.7**
>
> A **splay tree** is a binary search tree that, after every access (search, insertion, deletion), moves the accessed node to the root through a sequence of rotations, an operation called **splaying**. Nodes store no balance information.

A splay tree has no height guarantee and can even become a chain, whose bottom node costs *O(n)* to reach. But that access also roughly halves the depth of the nodes on its path. Sleator and Tarjan showed that any *m* operations from an empty tree take *O(m log n)* in total, an **amortized cost** of *O(log n)* per operation: the bound covers the whole sequence, with cheap operations paying for expensive ones.

The structure has three notable properties:

- **Locality:** Frequently accessed keys stay near the root, so the tree acts like a cache and can beat balanced trees when accesses concentrate on a few keys.
- **Simplicity:** No extra fields; one `splay` routine serves all operations, applied to the node found, the node inserted, or the node about to be deleted.
- **Drawback:** Even a search modifies the tree, a problem when several threads read it, and no single operation has a time guarantee, which real-time systems may not accept.

10.6.2 Zig, zig-zig and zig-zag

Splaying repeats three kinds of step until the accessed node x is the root; which step applies depends on its parent p and grandparent g.

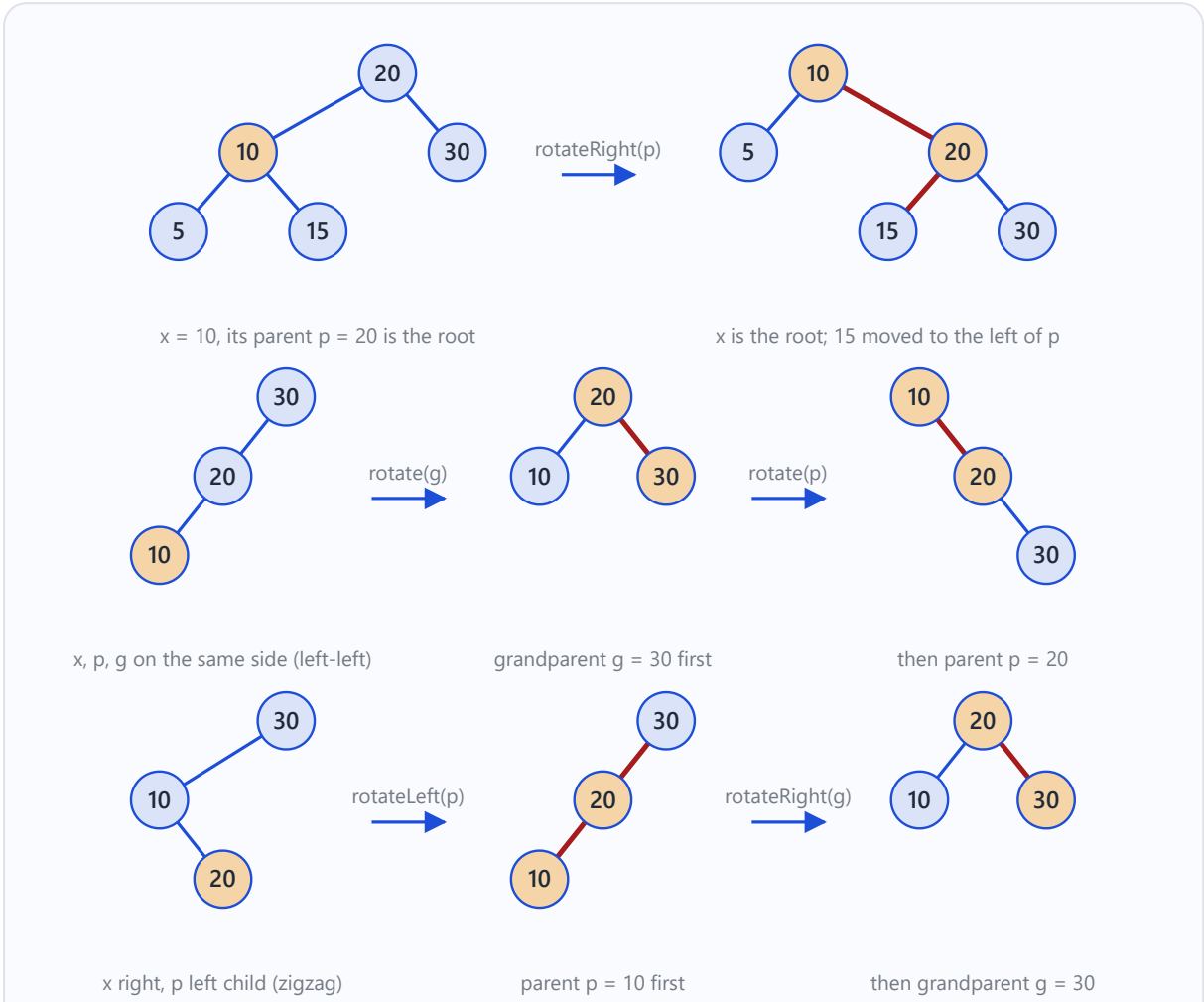


Figure 10.37: Splay steps. Zig: $x = 10$ under the root $p = 20$; one rotation, and 15 moves left of p. Zig-zig: x, p, g all left; rotate at 30, then 20. Zig-zag: x a right child, p a left child; rotate at 10, then 30.

- **Zig:** p is the root. One rotation at p makes x the root; this can only be the last step.
- **Zig-zig:** x and p are children on the same side. Rotate at the *grandparent* first, then the parent.
- **Zig-zag:** they are on opposite sides. Rotate at the parent, then the grandparent, as in an AVL double rotation.

The zig-zig order differs from AVL and matters. Rotating at the grandparent first "folds" the path, shortening the upper chain too; single rotations from the parent also bring x to the root but can leave the chain long. The amortized bound relies on the grandparent-first order.

Common mistake

Rotating the parent first in zig-zig. That is plain move-to-root, which lacks the amortized *O(log n)* guarantee: repeatedly accessing the bottom of a sorted chain keeps the chain long, and each access can cost *O(n)*. Rotate at *g* first, then *p*.

The slides' code puts the three steps in one loop:

```
void splay(Node x) {
    while (x != root) {
        Node p = x.parent, g = p.parent;
        if (g == null)                // zig
            rotate(p, x);
        else if (sameSide(x, p, g)) {  // zig-zig
            rotate(g, p);   // grandparent first
            rotate(p, x);
        } else {                      // zig-zag
            rotate(p, x);   // parent first
            rotate(g, x);
        }
    }
}
```

`rotate(a, b)` is the single rotation from the red-black code, lifting *b*, a child of *a*, into the place of *a*. `sameSide(x, p, g)` checks whether *x* hangs on the same side of *p* as *p* of *g*. The zig-zag branch ends with `rotate(g, x)` because after the first rotation *x* has replaced *p* as the child of *g*.

10.6.3 Code trace: splay(50) and splay(10)

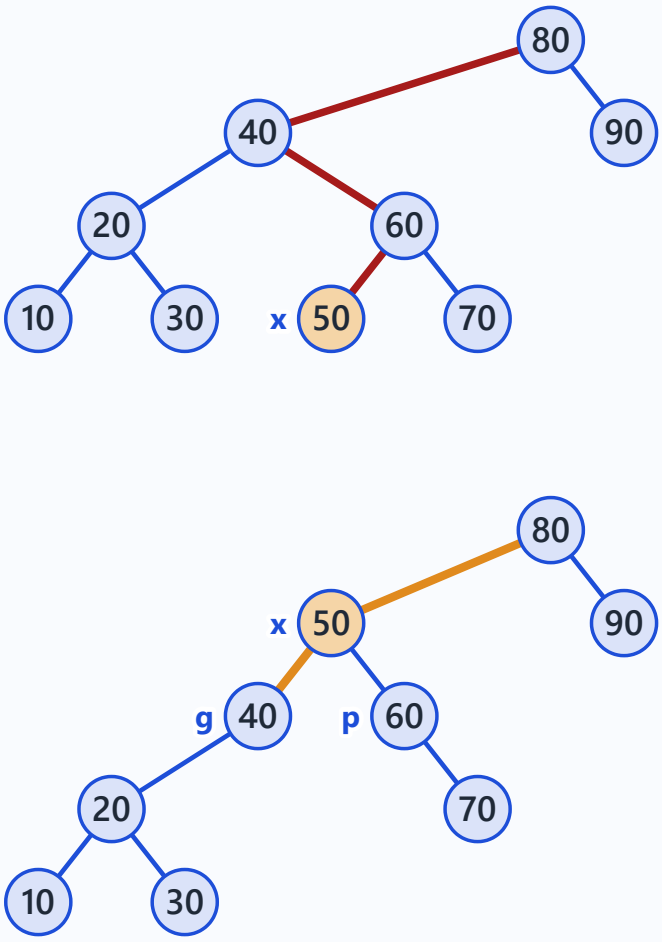
Start from 80(40(20(10, 30), 60(50, 70)), 90) and access 50, then 10.

Access	Iteration	*p*, *g*	Step	Rotations
50 (path 80 → 40 → 60 → 50)	1	60, 40	zig-zag (50 is a left child, 60 a right child)	right at 60, then left at 40
	2	80, none	zig	right at 80; 50 is the root
10 (path 50 → 40 → 20 → 10)	1	20, 40	zig-zig (both left children)	right at 40 first, then right at 20
	2	50, none	zig	right at 50; 10 is the root

Table 10.23: Trace of `splay(50)` followed by `splay(10)`.

After `splay(50)` the tree is 50(40(20(10, 30), —), 80(60(—, 70), 90)): 50 rose from depth 3 to the root, the old root 80 dropped a level taking 90 from depth 1 to 2, and 40, 20, 60, 70 kept their depths.

After `splay(10)` it is 10(—, 50(20(—, 40(30, —)), 80(60(—, 70), 90))). 10 is the root, and 50, accessed just before, is still at depth 1. Because zig-zig rotated at 40 first, 40 moved below 20 and took 30 as its left child.



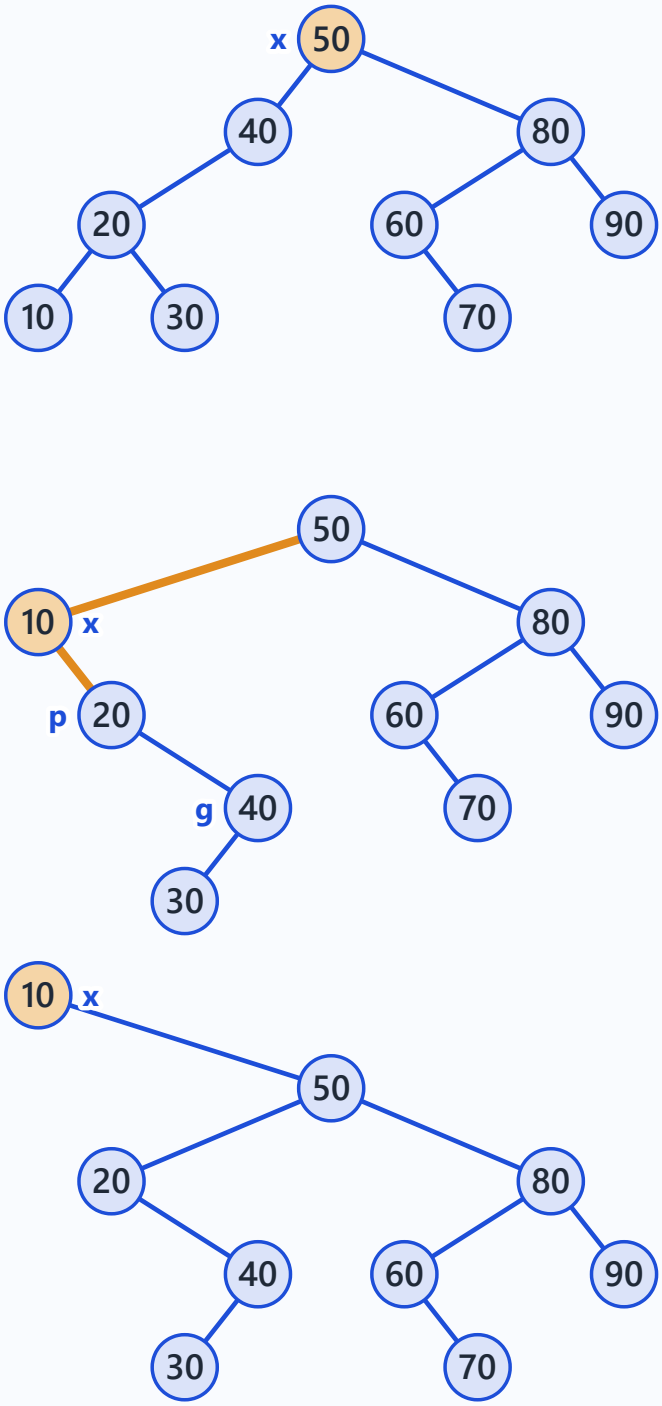


Figure 10.38: The splay trace. (a) `splay(50)` searches 80 → 40 → 60 → 50. (b) After the zig-zag, 50 is above 40 and 60. (c) After the zig, 50 is the root. (d) `splay(10)`, zig-zig: rotations at 40, then 20. (e) After the zig, 10 is the root; 50 is at depth 1.

One splay costs time proportional to the accessed node's depth, up to *O(n)*, but the constant reshaping keeps the average over long sequences at *O(log n)*.

10.7 Multiway search trees

So far every tree has been binary, which suits main memory, where following a link takes nanoseconds. On disk, reading a block is thousands of times slower than reading memory, hundreds of thousands on old spinning disks. A binary tree indexing a billion records has height at least 30; with one disk read per node, a search costs 30 accesses. Multiway trees cut this to a few.

10.7.1 The B-tree

Definition 10.8

A **B-tree** is a balanced multiway search tree whose nodes hold several keys and many children. In a B-tree of **order** (degree) m:

- A node holds at most m children and $m - 1$ keys.
- Every internal node other than the root has at least $\lceil m/2 \rceil$ children.
- Keys in a node are sorted. An internal node with k keys has $k + 1$ children, and the child between two adjacent keys holds the keys between them.
- All leaves are at the same depth.

A node's keys act as boundaries between its children. The root [20, 40] in Figure 10.39 sends keys below 20 left, keys between 20 and 40 to the middle, and larger keys right: the BST property generalized. A BST resembles a B-tree of order 2, but the B-tree's strength is the rule that all leaves share one depth, which keeps it perfectly balanced.

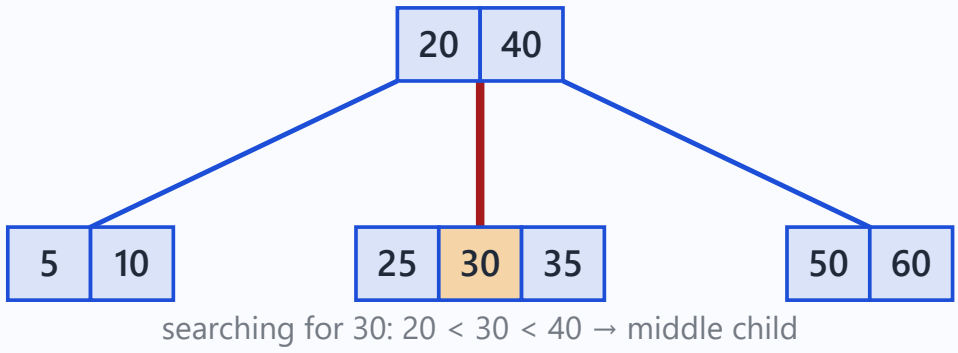


Figure 10.39: A B-tree of order 4: root [20, 40], leaves [5, 10], [25, 30, 35], [50, 60]. A search for 30 finds 20 < 30 < 40 at the root and goes to the middle child.

Why? Each node fills one disk block, which holds hundreds of keys. With order 200, a tree of height 3 holds 200^4, over a billion keys, and a search reads four blocks, fewer from disk since the root usually stays in memory. Being wide and shallow, B-trees, introduced by Bayer and McCreight in 1972, underlie database and file-system indexes.

A B-tree of order 4 is a 2-3-4 tree, whose nodes have 2, 3 or 4 children, and it is equivalent to a red-black tree: merging each black node with its red children yields a 2-3-4 tree, and "equal black height" becomes "all leaves at the same depth."

10.7.2 Search and insertion in a B-tree

Search generalizes BST search. At each node the keys are compared with the target; if it is not found, the search descends to the child for its interval. One node, one disk block, is read per level, so the cost is $O(\log_m n)$ reads; binary search within a node takes $O(\log m)$ comparisons, $O(\log n)$ in total.

Insertion always starts at a leaf: follow the search path and put the key in sorted position. If the leaf then reaches m keys, it **splits**: the median moves up to the parent and the remaining keys form two nodes. The parent, gaining a key and a child, may split in turn, and so on up to the root, whose split creates a new root and adds a level.

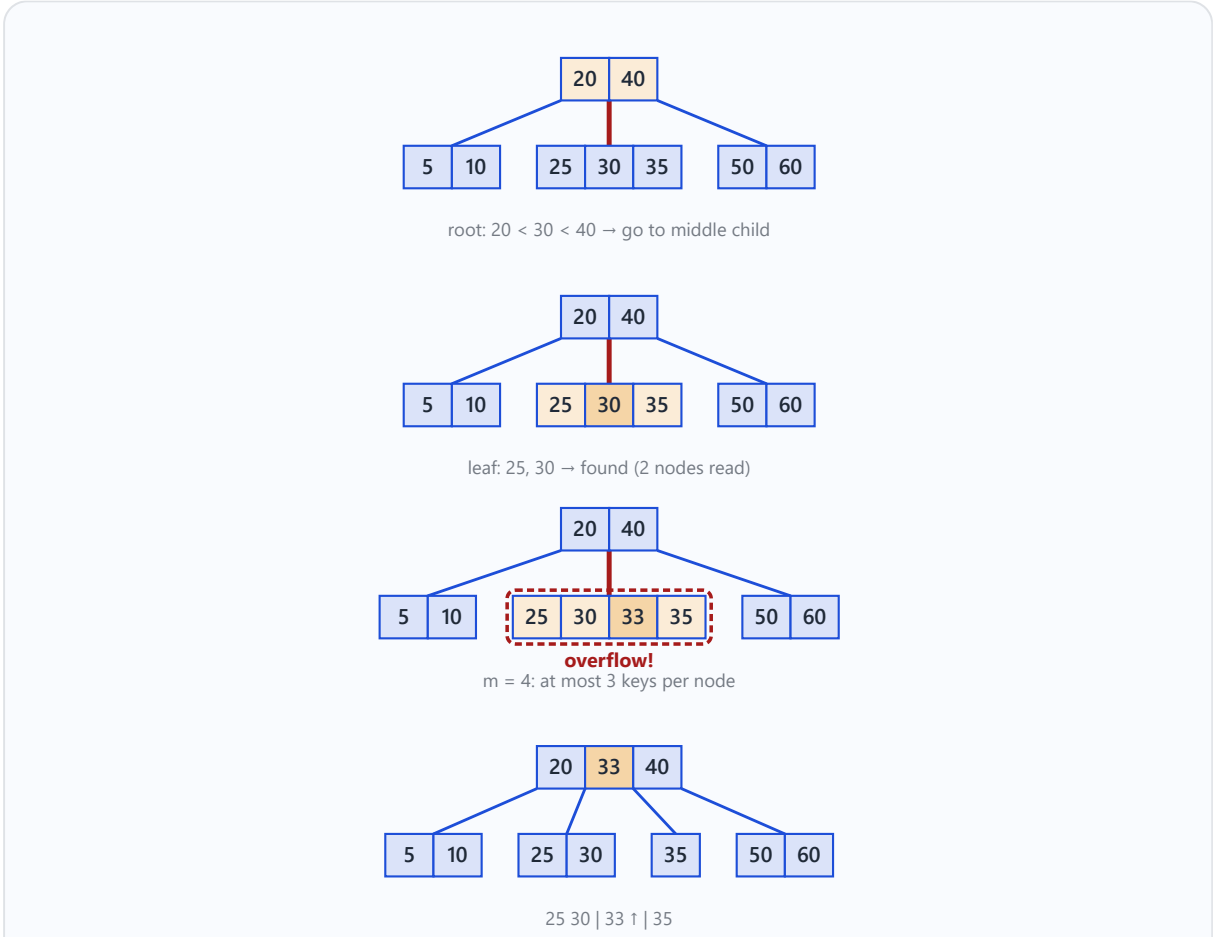


Figure 10.40: Search and insertion, order 4. (a) Searching for 30: 20 < 30 < 40, go to the middle child. (b) Found in the leaf; one node read per level. (c) 33 takes the same path into the leaf, which now has 4 keys, over

its capacity of $m - 1 = 3$. (d) The split sends 33 up, leaving [25, 30] and [35]; the root is now [20, 33, 40].

The four-key leaf [25, 30, 33, 35] has no single median. The slides pick the right middle key, 33, leaving two keys on the left and one on the right; the left one would also do, as long as the choice is consistent. Odd orders avoid the issue: with $m = 3$ an overflowing node has three keys and one median.

Splitting means a B-tree grows **from the bottom up**. A BST grows downward as leaves are added, but a B-tree gains height only when its root splits, pushing all leaves down at once, so they always stay at the same depth.

Deletion reverses insertion. A node that falls below minimum occupancy borrows a key from a sibling or merges with it; a merge takes a key from the parent, and the shortage may propagate upward. If the root loses its last key, it is removed and the height drops by one.

10.7.3 Insertion and splitting, step by step

Insert 10, 20, 30, 40, 50, 60, 70 into an empty B-tree of order $m = 3$, where a node holds at most 2 keys and 3 children and splits on reaching 3 keys.

Insert	Event	Result
10	added to the root leaf; 1 key	[10]
20	added to the root leaf; 2 keys	[10, 20]
30	[10, 20, 30] overflows; the median 20 moves up into a new root (root split, height 1)	[20]{[10] [30]}
40	added to the right leaf	[20]{[10] [30, 40]}
50	[30, 40, 50] overflows; 40 moves up to the parent, which has room	[20, 40]{[10] [30] [50]}
60	added to the right leaf	[20, 40]{[10] [30] [50, 60]}
70	[50, 60, 70] overflows; 60 goes up; the parent [20, 40, 60] overflows; 40 becomes the new root (root split, height 2)	[40]{[20]{[10] [30]} [60]{[50] [70]}}

Table 10.24: Inserting 10, 20, 30, 40, 50, 60, 70 into a B-tree with $m = 3$. Trees are written as "root{children}."

The 7 keys occupy 3 levels; every internal node has 2 children and all leaves share a depth. The height grew only at the two root splits. Sorted input that would turn a plain BST into a chain yields a perfectly balanced B-tree.

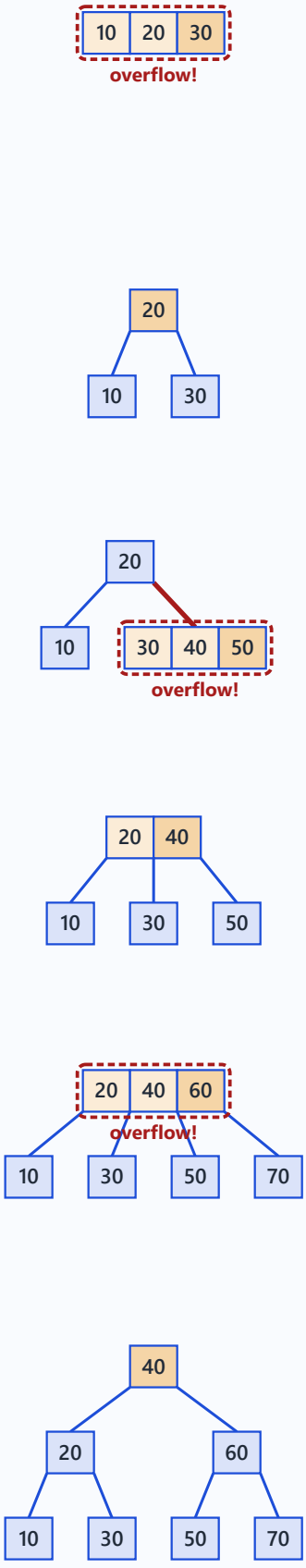


Figure 10.41: Splits. (a) 30 makes [10, 20, 30], over capacity. (b) Root split: 20 is the new root. (c) 50 overflows [30, 40, 50]. (d) 40 moves up; the split stops there. (e) 70 sends 60 up, overflowing the parent [20,

40, 60]. (f) Root split: 40 is the new root; one level taller, all leaves at the same depth.

Common mistake

Adding a new leaf below an overflowing one. A B-tree grows sideways: a split turns the full node into two siblings and sends the median up, and only a root split adds height. Nodes below the leaves would break the equal-depth rule.

10.7.4 The B+ tree

The B+ tree, the variant most used in databases, differs from the B-tree in three ways:

- **Data in the leaves:** All keys and records live in the leaves; internal nodes hold only routing keys, copies of leaf keys.
- **Linked leaves:** The leaves form a left-to-right linked list, so range queries like "20 to 50" walk along it.
- **Wider internal nodes:** Without records, more keys fit in a block, so the order rises and the tree gets shallower.

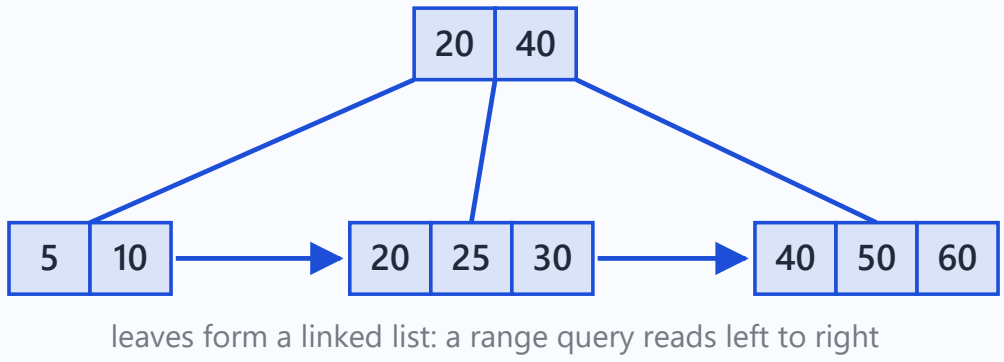


Figure 10.42: A B+ tree: routing keys 20 and 40 at the root; leaves [5, 10], [20, 25, 30], [40, 50, 60], linked for range queries. 20 and 40 appear at the root and in the leaves.

Every search ends at a leaf, where the records are; there is no B-tree-style early exit at an internal node. In return, every search descends the same number of levels, so its cost is predictable. Most relational database systems use B+ trees for table indexes.

10.7.5 Range queries in a B+ tree

Find the keys from 10 to 45 in the tree of Figure 10.42.

1. At the root, 10 < 20, so the leaf holding the lower bound is the leftmost child. This descent is a search, *O(log n)* levels.

2. Leaf [5, 10]: skip 5, take 10, then follow the link to the next leaf without going back up.
3. Leaf [20, 25, 30]: take all three and follow the link.
4. Third leaf [40, 50, 60]: take 40; stop at 50 > 45.

The result 10 20 25 30 40 is sorted. The cost is one descent plus the leaves read: $O(\log n + k)$ for k results. In a B-tree, with keys spread over internal nodes too, the query would climb up and down repeatedly; the leaf chain avoids that.

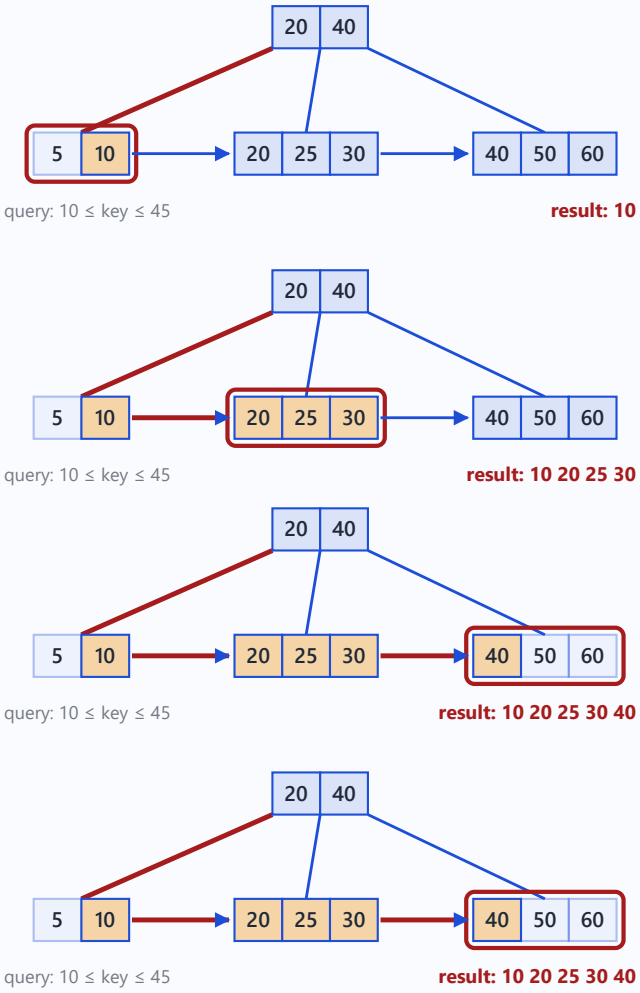


Figure 10.43: Range query 10 to 45 in a B+ tree. (a) First leaf: 10. (b) Second leaf: 20, 25, 30. (c) Third leaf: 40; stop at 50 > 45. (d) Result 10 20 25 30 40, cost $O(\log n + k)$.

Example 10.8

A registration system stores 2 million student records in a B+ tree of order 100, keyed by ID. Internal nodes are at least half full, so each has at least 50 children, and the height is at most about $\lceil \log_{50}(2 \cdot 10^6) \rceil$: since $50^3 = 125{,}000$ and $50^4 = 6{,}250{,}000$, at most 4 levels. Finding a student reads at most 4 blocks; listing the 150 students with IDs 2020001 to 2020150 takes

one descent and a few leaf reads. A balanced binary tree would have height at least 20.

10.8 Common mistakes

Most bugs in search tree code are of a few types. Those noted throughout the chapter are collected here as a checklist for reviewing tree code.

Mistake	What happens	Correct approach
Ignoring the return value	`delete(root, 40);` breaks the tree when the root changes.	`root = delete(root, 40);`, `root = insert(root, 5);`
Checking only children	Non-BSTs pass a parent–child check.	Carry a (min, max) interval down from the root (`isBST`).
Comparing objects with `==`	References are compared; a key in the tree is not found.	`compareTo` for order, `equals` or `compareTo(...) == 0` for equality.
Height update order	Wrong balance factors after an AVL rotation, needless rotations.	Update the node that moved down first, then the new root.
"A BST is always *O(log n)*"	On sorted input $h = n - 1$ and operations are *O(n)*.	*O(log n)* is guaranteed only in balanced trees (AVL, red-black).
Forgetting `null` children	`NullPointerException`, or wrong balance and color decisions.	Empty subtrees have height −1 and NIL is black; helpers like `h(null)` and `isRed(null)` encode this.
Single instead of double rotation	In LR or RL the tree tips the other way, still unbalanced.	Check the child's balance factor; in a zigzag, rotate the child first.
Zig-zig order	The splay tree loses its amortized guarantee.	Rotate the grandparent first, then the parent.

Table 10.25: Common mistakes in search tree code.

Two misconceptions lie behind these mistakes. One is seeing the tree only locally, a node with its parent and children, when the BST property, the AVL condition and the black-height rule all constrain whole subtrees. The other is forgetting Java's reference semantics: tree methods return a subtree's new root, and nothing changes until the caller stores it in a link or variable.

10.9 Summary

- A **search tree** keeps elements ordered by key; search, insertion and deletion follow one path from the root in *O(h)*, where h ranges from about $\log_2 n$ to $n - 1$.
- The **BST property**: all keys in a node's left subtree are smaller and all in its right subtree larger, for whole subtrees. **Inorder traversal** gives sorted output in *O(n)*. The minimum is leftmost, the maximum rightmost.
- **Insertion** descends as in a search and attaches a leaf at the empty link reached. Methods return the subtree's new root: `root = insert(root, x)`.

- The **successor** of x is the smallest larger key, the **predecessor** the largest smaller one; ceiling and floor also accept equality. All take $O(h)$.
- **Deletion** has three cases: remove a leaf; bypass a one-child node; replace a two-child node's value with its successor's (or predecessor's) and delete that node, which, having no left child, is an easy case.
- A **range query** with pruning takes $O(h + s)$. **BST validation** carries an allowed interval down from the root and takes $O(n)$.
- A plain BST's shape follows insertion order (sorted input gives a chain; random order, average depth about $1.39 \log_2 n$). **Balanced trees** guarantee $h = O(\log n)$ by small local changes.
- An **AVL tree** has $bf = h(\text{left}) - h(\text{right}) \in \{-1, 0, 1\}$ at every node. The sparsest AVL tree grows like the Fibonacci numbers, so $h < 1.44 \log_2(n + 2)$. LL and RR need one **rotation**, LR and RL two; a rotation is $O(1)$ and preserves the inorder sequence.
- A **red-black tree** has a black root, no adjacent reds, and equal black counts on all paths, so the longest path is at most twice the shortest and $h \le 2 \log_2(n + 1)$. Insertion fix-up depends on the uncle: red means recolor, black means at most two rotations.
- A **splay tree** moves each accessed node to the root by zig, zig-zig and zig-zag steps: $O(n)$ worst case per operation, $O(\log n)$ amortized.
- A **B-tree** node holds at most $m - 1$ keys and m children, and all leaves share a depth. Overflowing nodes split, sending the median up, so the tree grows from the bottom. A **B+ tree** keeps data only in linked leaves; range queries take $O(\log n + k)$.

Structure	Height	Search / insert / delete	How is it balanced?
BST	$\log n \dots n$	$O(h)$	not balanced; worst case $O(n)$
AVL	$\le 1.44 \log n$	$O(\log n)$	$bf \in \{-1, 0, 1\}$; LL, RR, LR, RL rotations
Red-black	$\le 2 \log(n + 1)$	$O(\log n)$	recoloring + at most 2–3 rotations
Splay	unbounded	$O(\log n)$*	accessed node moved to the root by zig / zig-zig / zig-zag
B-tree / B+	$\approx \log_m n$	$O(\log n)$	overflowing nodes split; all leaves at the same depth

Table 10.26: Operation costs. (*) Amortized; one operation can be $O(n)$. Inorder traversal is always $O(n)$ and sorted.

10.10 Key terms

Each term links to the course glossary with its Turkish equivalent.

search tree · binary search tree (BST) · key · root · leaf · subtree · depth · height · inorder · traversal · successor · predecessor · ancestor · sibling · perfect binary tree · balance factor · AVL tree · rotation · red-black tree · splay tree · zig, zig-zig, zig-zag · amortized analysis · B-tree · B+ tree · degree

10.11 Test Yourself

1. Insert 35, 18, 52, 10, 25, 44, 60, 22, 48 into an empty BST. What is its height and its inorder traversal? How many nodes does `search(root, 48)` compare?

   Answer

   The tree is 35(18(10, 25(22, —)), 52(44(—, 48), 60)). The deepest nodes, 22 and 48, are at depth 3, so the height is 3. Inorder: 10 18 22 25 35 44 48 52 60. Searching for 48 compares 35 (right), 52 (left), 44 (right) and 48: 4 nodes, the depth of 48 plus one.

2. Delete 18 and then 35 from the same tree, using the inorder successor for two-child nodes. What is the tree after each deletion?

   Answer

   The successor of 18 is 22 (right to 25, then left); 22 is copied into the node of 18 and the old leaf deleted: 35(22(10, 25), 52(44(—, 48), 60)). The successor of 35, left from 52, is 44; it is copied to the root and the old 44, having one child, is bypassed: 44(22(10, 25), 52(48, 60)).

3. In the tree of question 1, what are `successor(25)`, `predecessor(44)`, `ceiling(45)` and `floor(21)`?

   Answer

   `successor(25) = 35` and `predecessor(44) = 35`: 25 has no right subtree and 44 no left subtree, and in both cases the nearest qualifying ancestor is the root. `ceiling(45) = 48` and `floor(21) = 18`; 45 and 21 are not in the tree.

4. In the tree of question 1, what does `range(root, 20, 46)` print? Which nodes are never visited?

   Answer

   22 25 35 44, visiting 35, 18, 25, 22, 52, 44, 48. At 18, `20 < 18` is false, pruning 10; at 52, `52 < 46` is false, pruning 60. 48 is visited (right of 44, since 44 < 46) but is out of range. 10 and 60 are never visited.

5. Insert 30, 20, 10, 40, 50, 35 into an empty AVL tree. Which insertions trigger which case, and what

is the final tree?

Answer

10: *bf(30) = 2*, *bf(20) = 1* → LL; right at 30 gives 20(10, 30). 40: balanced. 50: *bf(30) = −2*, *bf(40) = −1* → RR; left at 30 gives 20(10, 40(30, 50)). 35 lands right of 30: *bf(20) = −2*, *bf(40) = 1* → RL; right at 40, then left at 20. Final tree 30(20(10, —), 40(35, 50)), height 2.

6. Insert 30, 20, 10, 25, 27 into an empty red-black tree. Which case applies at each insertion needing a fix-up? Give the final tree with colors.

Answer

10: red parent 20, NIL uncle, a line → right at 30; 20B(10R, 30R). 25: red parent 30, red uncle 10 → recolor (10, 30 black, 20 red), and the root 20 turns black again. 27: red parent 25, NIL uncle, a triangle (27 right, 25 left child) → left at 25, right at 30; 27 black, 30 red. Final tree 20B(10B, 27B(25R, 30R)), 2 blacks on every path.

7. Given the splay tree 50(30(20(10, —), 40), 70), which step does `splay(40)` use, and what is the result?

Answer

The path is 50 → 30 → 40 (p = 30, g = 50); 40 is a right child, 30 a left child: zig-zag. Rotating left at 30 and right at 50 gives 40(30(20(10, —), —), 50(—, 70)) in one step.

8. Insert 5, 15, 25, 35, 45 into an empty B-tree of order 3. Which insertions cause splits, and what is the final tree?

Answer

25 overflows [5, 15, 25], and 15 moves up into a new root: [15]{[5] [25]}. 35 joins the right leaf: [25, 35]. 45 overflows [25, 35, 45]; 35 moves up to the root, which has room: [15, 35] {[5] [25] [45]}. Two splits, one at the root; height 1.

9. In a red-black tree whose root has black height 3, how many nodes can the shortest and longest root-to-NIL paths contain? What is the minimum number of nodes in such a tree?

Answer

Every path has 3 black nodes. The shortest is all black: 3 nodes. On the longest, a red can follow each black: at most 6 (black, red, black, red, black, red, as the root is black). The minimum is $2^3 - 1 = 7$ nodes: the all-black perfect tree of height 2.

10. In the B+ tree of Figure 10.42, which keys lie between 22 and 55, and how many leaves are read?

Answer

At the root, 20 ≤ 22 < 40 leads to the middle leaf [20, 25, 30]: skip 20, take 25 and 30. The link leads to [40, 50, 60]: take 40 and 50, stop at 60 > 55. Result 25 30 40 50; 2 leaves read, with no return to the root.

10.12 Further reading

- M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, *Data Structures and Algorithms in Java* (6th ed., 2014), Chapter 11: covers binary search trees, AVL, splay, (2, 4) and red-black trees in the same order, with Java code close to that of these notes.
- R. Sedgewick and K. Wayne, *Algorithms* (4th ed., 2011), Sections 3.2 and 3.3: presents BSTs and left-leaning red-black trees by relating them to 2-3 trees; the easiest way to see why red-black trees work.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms* (4th ed., 2022), Chapters 12, 13 and 18: BSTs, red-black trees (with all cases, including deletion) and B-trees, with proofs.
- D. D. Sleator and R. E. Tarjan, "Self-Adjusting Binary Search Trees" (1985): the paper that introduced splay trees, proving the amortized *O(log n)* bound with a potential function.

10.13 References

1. G. M. Adelson-Velsky, E. M. Landis, "An algorithm for the organization of information", *Soviet Mathematics Doklady*, 3, 1259–1263, 1962.
2. R. Bayer, E. M. McCreight, "Organization and maintenance of large ordered indexes", *Acta Informatica*, 1(3), 173–189, 1972.
3. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022.
4. M. T. Goodrich, R. Tamassia, M. H. Goldwasser, *Data Structures and Algorithms in Java*, 6th ed., Wiley, 2014.
5. L. J. Guibas, R. Sedgewick, "A dichromatic framework for balanced trees", *19th Annual Symposium on Foundations of Computer Science*, 8–21, 1978.
6. R. Sedgewick, K. Wayne, *Algorithms*, 4th ed., Addison-Wesley, 2011.
7. D. D. Sleator, R. E. Tarjan, "Self-adjusting binary search trees", *Journal of the ACM*, 32(3), 652–686, 1985.

PART III · CHAPTER 11

Priority Queues and Heaps

An ordinary queue serves whoever came first. Real life rarely works that way: in an emergency room, a heart attack is seen before the sprained ankle that arrived earlier. A priority queue captures this rule. We define it abstractly and implement it first with an unsorted list, then with a sorted one; each leaves one operation slow. The binary heap balances the two, inserting and removing in logarithmic time. Its array representation, up-heap and down-heap, key changes, linear-time construction, and heap sort are worked through with code traces. The chapter closes with Java's `PriorityQueue`, the kth-largest-element problem, and priority scheduling in operating systems.

Code for this chapter: `github.com/sercankulcu/data-structures-java/Bolum11/src`

LEARNING OUTCOMES

The ADT
Explain the operations and uses of a priority queue; trace a sequence of operations.

Binary heaps
Relate heap order and completeness to the array representation; compute parent and child indices.

Operations
Carry out insertion, root removal, key changes, and heap construction by hand and in code.

Cost analysis
Justify each operation's cost; explain why bottom-up construction is linear and how heap sort works.

11.1 The priority queue ADT

A data structure can be described at two levels: *what* it does (its operations, and what each changes and returns) and *how* it does it (the memory layout and the steps of each operation). A description at the first level is an **abstract data type** (ADT), and the priority queue is one. The same ADT can be implemented by very different data structures, and a main lesson of this chapter is how much that choice affects running time.

11.1.1 What is a priority queue?

Picture a traffic jam. Cars move forward in the order they arrived, until an ambulance comes up behind with its siren on. Drivers pull aside and it passes them all, not because it arrived early but because its priority is high.

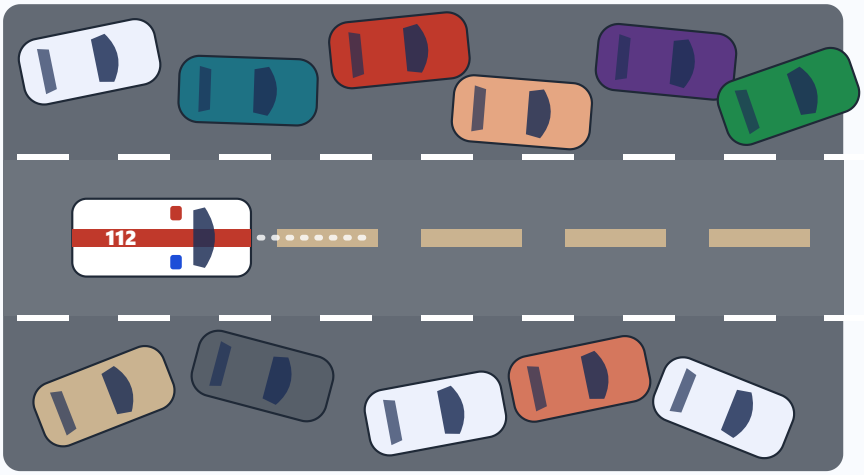


Figure 11.1: In a traffic jam, cars pull aside and the ambulance passes through the opened lane. The highest priority goes first, not the earliest arrival.

Computer systems are full of this pattern (Figure 11.1). An operating system runs the most important waiting process; a print queue can move an administrator's urgent job to the front; an event simulation handles the earliest event next. In each case the waiting items carry a priority, and the next item is chosen by it.

Definition 11.1

A **priority queue** is an abstract data type whose items each carry a priority and whose removal operation always returns an item of highest priority. The order of arrival does not determine the order of departure; the priorities do.

Note what the definition does not demand. A priority queue must answer only one question quickly, namely which item matters most right now; it need not keep its items sorted. That freedom is what makes efficient implementations possible.

Costs depend on the implementation. In an unsorted list, even looking at the top item means scanning everything, *O(n)*. In a binary heap, looking takes *O(1)* and removing *O(log n)*.

11.1.2 Keys, values, and priorities

Each item has two parts: a **key**, which carries its priority, and a **value**, the data actually stored. In the emergency room, `(1, "heart attack")` and `(4, "sprain")` are two items; the first has key 1.

Definition 11.2

Each item in a priority queue is a **(key, value)** pair. Keys must be comparable: for any two keys, $<$ and $\leq$ decide which is smaller. In a **min-priority queue** a smaller key means higher priority, and removal returns an item with the smallest key; in a **max-priority queue** the largest key wins.

Which direction to use is a convention. Goodrich, Tamassia, and Goldwasser define the ADT as a min-priority queue with `min()` and `removeMin()`, and so do these notes, while most of the heap code uses a max-heap with `insert` and `removeMax`. The two differ only in the direction of comparison: change every $<$ to $>$ and a min-queue becomes a max-queue, so everything carries over.

The **highest-priority item** is the one at the "front": the smallest key in a min-queue, the largest in a max-queue. If several items share that key, any of them counts as the front.

So **in what order do equal-priority items leave?** The ADT makes no promise. Nothing guarantees FIFO (first-in, first-out) order among equal keys; some implementations happen to preserve it, others don't, and Java's `PriorityQueue` gives no guarantee.

If ties must go by arrival, add an **arrival number**, taken from a counter, to the key. Comparison looks at the priority first and, on a tie, at the number, so the earlier item wins. Since the numbers are distinct, no two keys are ever equal and the order is fully determined.

Common mistake

Don't assume that a priority queue keeps insertion order among equal keys. Heap-based implementations move items around with swaps, so equal keys can leave in any order. For FIFO behavior, use a (priority, arrival number) key.

11.1.3 Queue or priority queue?

A small example shows the difference.

Example 11.1

Four patients arrive in this order; a smaller number is more urgent: Alan (3), Eve (1), Carl (2), Nina (1). In what order does an ordinary queue serve them, and in what order a min-priority queue?

1. **Queue (FIFO).** In arrival order: Alan, Eve, Carl, Nina. Priorities are ignored.
2. **Priority queue.** Smallest key first. Eve and Nina both have key 1; keeping arrival order, Eve goes first, then Nina.

3. Then Carl (2) and finally Alan (3): Eve, Nina, Carl, Alan.

Order	1st	2nd	3rd	4th
Arrival (priority)	Alan (3)	Eve (1)	Carl (2)	Nina (1)
Queue (FIFO)	Alan	Eve	Carl	Nina
Priority queue	Eve	Nina	Carl	Alan

Table 11.1: The same four patients leaving a queue and a priority queue. Eve and Nina have equal priority; here arrival order is kept.

Alan arrived first but leaves the priority queue last. The relative order of Eve and Nina is the implementation's choice, not the ADT's; another implementation could send Nina out first.

A priority queue generalizes the ordinary queue: with arrival time as the key, a min-priority queue returns the oldest item, like a queue; with negated arrival time, it returns the newest, like a stack.

11.1.4 Applications

A priority queue fits any problem of the form “repeatedly take the best of what is waiting.” Typical areas:

- **Operating systems.** The highest-priority process gets the CPU first: an interrupt handler before the user interface, the interface before a background backup.
- **Graph algorithms.** Dijkstra's shortest-path algorithm and A* search take the next closest node from a priority queue, keyed by its best known distance from the source.
- **Emergency management.** Triage and incident handling deal with the most urgent case first.
- **Data compression.** Huffman coding repeatedly merges the two lowest-frequency nodes, taking them with two `removeMin` calls and inserting the merged node back.

In all of these, items arrive over time and removals are interleaved with insertions. If all the data were known in advance, one sort would do. In Dijkstra's algorithm, though, each removal causes new insertions, and keeping a sorted array up to date after every insertion is expensive. Priority queues are built for exactly this.

11.1.5 Operations and a Java interface

The min-priority queue ADT has the five operations in Table 11.2.

Operation	Meaning
`insert(k, v)`	Inserts an item with key `k` and value `v`.
`removeMin()`	Removes and returns an item with the smallest key.
`min()`	Returns an item with the smallest key without removing it (peek).

Operation	Meaning
`size()`	Returns the number of items.
`isEmpty()`	Returns `true` if the queue is empty.

Table 11.2: The operations of a min-priority queue.

In Java, an ADT's contract is naturally an interface, which gives each operation's name, parameters, and return type but not its implementation:

```
public interface PriorityQueue<K, V> {
    void insert(K key, V value);
    Entry<K, V> min();        // null if empty
    Entry<K, V> removeMin();  // null if empty
    int size();
    boolean isEmpty();
}
```

`K` and `V` are the key and value types. `Entry<K, V>` is a small (key, value) pair type, returned by `min()` and `removeMin()` so that the caller gets the key too. The comments fix the empty-queue behavior: both methods return `null`. Some libraries throw an exception instead; either way, the choice is part of the contract that client code must know.

A max-priority queue calls these operations `max()` and `removeMax()`; the max-heap code in these notes uses `insert` and `removeMax`.

The interface says nothing about cost; that depends on the implementing class. An unsorted list, a sorted list, and a heap can all implement `PriorityQueue<K, V>`. Client code stays the same, and only the running time changes.

11.1.6 A sequence of operations

Table 11.3 applies a sequence of operations to an empty min-priority queue. Contents are written as sets, since the ADT defines no internal order; items are listed by key for readability.

Operation	Returns	Contents
`insert(5, A)`	—	{ (5,A) }
`insert(9, C)`	—	{ (5,A), (9,C) }
`insert(3, B)`	—	{ (3,B), (5,A), (9,C) }
`min()`	(3,B)	{ (3,B), (5,A), (9,C) }
`removeMin()`	(3,B)	{ (5,A), (9,C) }
`insert(7, D)`	—	{ (5,A), (7,D), (9,C) }
`removeMin()`	(5,A)	{ (7,D), (9,C) }
`removeMin()`	(7,D)	{ (9,C) }

Operation	Returns	Contents
`removeMin()`	(9,C)	{ }
`removeMin()`	`null`	{ } (empty queue)

Table 11.3: A sequence of operations on an empty min-priority queue.

Note three things. `min()` and `removeMin()` return the same item, but only the latter removes it. (3,B) entered after (5,A) and (9,C) yet leaves first, and (7,D), inserted last, leaves before (9,C): the key decides. Finally, `removeMin()` on an empty queue returns `null`, as the contract says.

Common mistake

`min()` (`peek()` in Java) does not remove the item. It only looks, and two calls in a row return the same item. Removal is `removeMin()` (`poll()` in Java).

11.2 List-based implementations

The simplest implementation keeps the items in a list, either unsorted, in arrival order, or always sorted by priority. Both work, but their costs are mirror images: one makes insertion cheap and removal expensive, the other the reverse. Comparing them in code shows why we need the binary heap.

11.2.1 Unsorted and sorted lists

An **unsorted list** appends each new item without any comparisons, so insertion is *O(1)*. Removal pays: the smallest key could be anywhere, so `min` and `removeMin` look at every item, *O(n)*.

A **sorted list** keeps the items in key order, so the smallest is always at one end and `min` and `removeMin` take *O(1)*. Now insertion pays: the new item's place must be found and items shifted, *O(n)* in the worst case.



Figure 11.2: Left, the unsorted list 5, 9, 3, 7: the new item 4 is appended, and finding the minimum scans

everything. Right, the sorted list: 4 goes between 3 and 5, and the minimum is always first.

Each implementation makes one operation fast and the other slow (Figure 11.2). Applications usually insert and remove about equally often, so the slow operation dominates, and n insertions plus n removals cost $O(n^2)$ either way. The binary heap balances the two at $O(\log n)$ each.

Which way to sort is an implementation choice. The figure puts the minimum first, but an array should keep it at the *end*: deleting the last cell of an array is cheap, as is deleting the first node of a linked list.

11.2.2 Code trace: removeMin() on an unsorted array

In an array, the items occupy `a[0]` through `a[n - 1]`. Insertion is just `a[n] = x; n++;`, and removal looks like this:

```
// unsorted array: insert(x) { a[n] = x; n++; }  O(1)
public int removeMin() {        // requires n > 0
  int minIdx = 0;               // index of the minimum
  for (int k = 1; k < n; k++)
    if (a[k] < a[minIdx]) minIdx = k;
  int min = a[minIdx];
  a[minIdx] = a[n - 1];  n--;   // last item fills the gap
  return min;
}
```

The code first finds the minimum: `minIdx` holds the index of the smallest item seen so far, starting at 0, and the loop updates it whenever `a[k]` is smaller. It then saves that item in `min`.

Finally it closes the gap. With no order to preserve, there is no need to shift everything left; moving the last item into the gap and decrementing `n` is a single $O(1)$ assignment. On a sorted array this trick would break the order.

Table 11.4 traces it on `a = [5, 9, 3, 7, 4]`, `n = 5`.

Step	Comparison	`minIdx`	Array
start	—	0	[5, 9, 3, 7, 4], `n = 5`
`k = 1`	`a[1] = 9 < a[0] = 5` ? no	0	unchanged
`k = 2`	`a[2] = 3 < a[0] = 5` ? **yes**	2	unchanged
`k = 3`	`a[3] = 7 < a[2] = 3` ? no	2	unchanged
`k = 4`	`a[4] = 4 < a[2] = 3` ? no	2	unchanged
`k = 5 = n`	loop ends	2	4 comparisons made
`min = a[2]`	—	2	`min = 3`
`a[2] = a[4]; n--`	—	2	[5, 9, 4, 7], `n = 4`

Step	Comparison	`minIdx`	Array
`return`	—	—	returns 3

Table 11.4: Trace of `removeMin()` on `a = [5, 9, 3, 7, 4]`.



Figure 11.3: `removeMin()` on an unsorted array. (a) First candidate: `minIdx = 0`. (b) At `k = 2`, 3 is smaller: `minIdx = 2`. (c) The last item, 4, fills the gap; `n = 4`. (d) 3 is returned; [5, 9, 4, 7] remains unsorted.

Five items took 4 comparisons. In general the loop runs $n - 1$ times wherever the minimum is, since even a minimum in the first cell can't be confirmed until every item has been seen. So `removeMin` is $\Theta(n)$ in every case, and `min()` does the same scan without the final move.

The result, [5, 9, 4, 7], is unsorted, which is fine; but note that the move has rearranged it, carrying 4 from the end to the middle.

Common mistake

Closing the gap in an unsorted array by shifting everything after it one place left is correct but wastes $O(n)$ work. Since order doesn't matter, move the last item into the gap. On a sorted array, of course, you can't.

11.2.3 Code trace: insert(4) on a sorted array

A sorted array must choose which end holds the minimum. At `a[0]`, every removal shifts all the other items left, making `removeMin` $O(n)$ and losing the sorted version's only advantage. So keep the array in **decreasing** order: the minimum is then at `a[n - 1]`, and removing it just decrements `n`:

```
// sorted array: decreasing order, min always at a[n-1]
public void insert(int x) {
  int k = n - 1;
  while (k >= 0 && a[k] < x) {    // smaller than x
    a[k + 1] = a[k];              // shift one right
    k--;
  }
  a[k + 1] = x;  n++;
}
public int removeMin() { n--; return a[n]; }
```

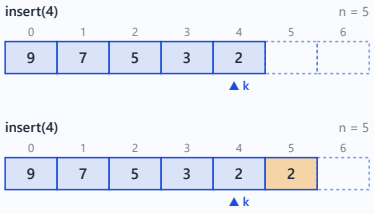
`insert` searches from the back. While `a[k] < x`, the item `a[k]` belongs after `x` in decreasing order, so it shifts right to `a[k + 1]` and `k` decreases. The loop stops when `k` reaches −1 (`x` is the largest) or `a[k] ≥ x`; either way `x` goes at `k + 1`. The test `k >= 0` comes first so that, by short-circuit evaluation of `&&`, `a[-1]` is never read.

Here is `insert(4)` on `a = [9, 7, 5, 3, 2]`, `n = 5`.

Step	k	Condition	Array
start	4	—	[9, 7, 5, 3, 2, ·]
compare	4	`a[4] = 2 < 4`: true	[9, 7, 5, 3, 2, ·]
`a[5] = a[4]`	4 → 3	—	[9, 7, 5, 3, 2, 2]
compare	3	`a[3] = 3 < 4`: true	[9, 7, 5, 3, 2, 2]
`a[4] = a[3]`	3 → 2	—	[9, 7, 5, 3, 3, 2]
compare	2	`a[2] = 5 < 4`: false, loop ends	[9, 7, 5, 3, 3, 2]
`a[3] = 4; n++`	2	—	[9, 7, 5, 4, 3, 2], `n = 6`
`removeMin()`	—	`n--`, returns `a[5]`	returns 2, `n = 5`

Table 11.5: Trace of `insert(4)` followed by `removeMin()` on `a = [9, 7, 5, 3, 2]`. The first six cells of the array are shown; an empty cell is written "·".

Two items (2 and 3) shifted right. In between, the array briefly holds duplicates: after `a[5] = a[4]`, 2 occupies two cells. That is not a bug. The extra copy is overwritten by 3 in the next step, and the leftover 3 is finally overwritten by 4. Shifting copies each item one cell right and needs fewer assignments than swapping.



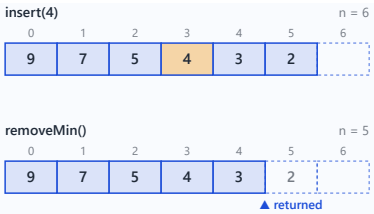


Figure 11.4: `insert(4)` on a sorted array. (a) Start at `k = n − 1 = 4`. (b) 2 has shifted right; `a[4]` is a leftover copy. (c) 4 is written to `a[3]`, `n = 6`. (d) `removeMin()` returns the final 2 with no shifting.

The cost is proportional to the number of items shifted: none if the new item is the smallest, all *n* if it is the largest, and about *n/2* on average, so insertion is *O(n)*. `removeMin()` is one line and *O(1)*, because the minimum is already at the end.

Common mistake

Keeping the highest-priority item at `a[0]` of a sorted array forces every `removeMin` to shift the rest left, so both operations become *O(n)*. Keep it at the *end*, and removal is *O(1)*.

11.2.4 Arrays versus linked lists

The sorted implementation can also use a linked list. The costs are similar but arise differently.

In an **array**, keeping the items sorted means shifting on insertion, *O(n)*. The highest-priority item is kept at the end, so removal is *O(1)*.

A **linked list** needs no shifting, but finding the new node's place still takes *O(n)*, since a position can only be reached by walking from the head; linking the node in is then two *O(1)* assignments. Here the highest-priority item is kept at the *head*: deleting the head is one assignment, while deleting the tail of a singly linked list means finding the next-to-last node. Either way, the top item sits at the cheap end. The linked list pays in memory, since each node also stores a reference to the next.

Example 11.2

Insert 6 into the increasing list 3 → 5 → 7 → 9. The list is min-priority, so the top item, 3, is at the head.

1. **Create.** A node `newNode` with value 6 is created, not yet linked.
2. **Search.** `prev` starts at the head and advances while `prev.next.value` is less than 6: 5 < 6, advance; 7 > 6, stop. `prev` stays at 5. In the worst case the scan walks the whole list: *O(n)*.
3. **First link.** `newNode.next = prev.next` : 6 points to 7.
4. **Second link.** `prev.next = newNode` : 5 points to 6, giving 3 → 5 → 6 → 7 → 9.

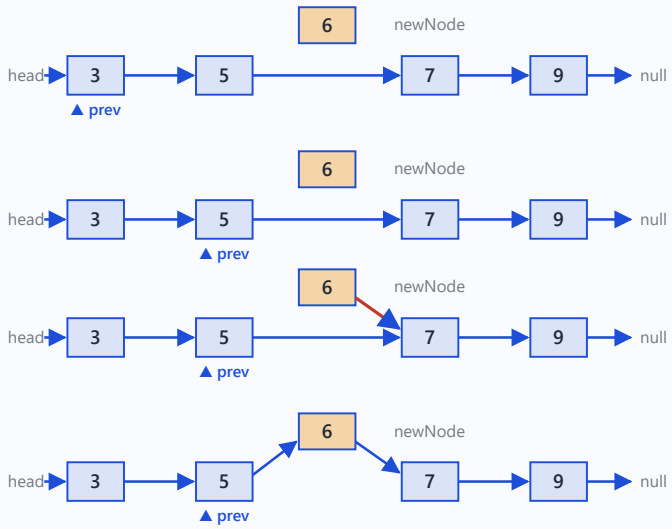


Figure 11.5: Inserting 6 into a sorted linked list. (a) The new node. (b) The search stops at `prev = 5`. (c) `newNode.next = prev.next` (6 → 7). (d) `prev.next = newNode` (5 → 6).

The order of the two links matters. Doing `prev.next = newNode` first would destroy the only reference to 7, and the rest of the list would be lost. Link the new node to the rest of the list first, then the list to the new node.

11.2.5 Code: a priority queue on a sorted array

Next, the sorted array becomes a general class whose items carry data and a priority. It is max-priority: larger priorities leave first. Each item is a small object:

```
class Item<T> {
    T data;
    int priority;
    Item(T v, int p) { data = v; priority = p; }
}
```

`Item<T>` is generic: `data` can be of any type, and `priority` is an integer. The queue keeps the items in an array:

```
// fields of SortedArrayPQ<E>
private Item<E>[] items =
    (Item<E>[]) new Item[10];
private int size = 0;
public boolean isEmpty() { return size == 0; }
```

The capacity starts at 10, and `size` counts the items. Java can't create generic arrays, so the code creates a raw `Item` array and casts it; the compiler's "unchecked cast" warning is harmless here because the array never leaves the class. Insertion:

```
public void insert(E data, int p) {
    if (size == items.length)
        items = Arrays.copyOf(items, 2 * size);
    int i = size;
    while (i > 0 && items[i-1].priority < p) {
        items[i] = items[i - 1];  // shift right
        i--;
    }
    items[i] = new Item<>(data, p);
    size++;
}
```

If the array is full, `Arrays.copyOf` makes one twice as large. Then `i` starts at the first empty cell, `size`, and moves left, shifting right every item whose priority is less than `p`. Where the loop stops, the new item goes. The largest priority thus sits at `items[0]`, where removal takes it:

```
public E remove() {
    if (isEmpty()) return null;
    E result = items[0].data;
    for (int i = 1; i < size; i++)
        items[i - 1] = items[i];
    items[--size] = null;
    return result;
}
```

`remove` returns `null` on an empty queue; otherwise it takes the data from `items[0]` and shifts the rest left. Writing `null` into the vacated last cell drops a stale reference that would keep the removed object from being garbage-collected. `items[--size]` decrements `size` before indexing.

This class ignores the advice to keep the top item at the end. It sits at the front, so `remove` is *O(n)*, as is `insert` in the worst case. The layout was chosen for readability; with the top at the end, `remove` would be one line.

The strict `<` in the loop matters too. An item of equal priority is not shifted, so the new item lands behind it. Equal-priority items therefore stay, and leave, in arrival order: the queue is FIFO on ties.

11.2.6 Code trace: inserting with equal priority

After `insert(A, 2)`, `insert(B, 5)`, `insert(C, 1)`, the array is `[B:5, A:2, C:1]` (cells written "data:priority"). Now `insert(D, 5)` is called. D ties with B: does it go in front of B or behind it?

```
public void insert(E data, int p) {
  if (size == items.length)
    items = Arrays.copyOf(items, 2 * size);
  int i = size;
  while (i > 0 && items[i-1].priority < p) {
    items[i] = items[i - 1];  // shift right
    i--;
  }
  items[i] = new Item<>(data, p);
  size++;
}
```

Step	`i`	Condition	`items`
capacity check	—	`3 == 10` ? no	[B:5, A:2, C:1, ·]
`i = size`	3	—	[B:5, A:2, C:1, ·]
compare	3	`items[2].priority = 1 < 5` : true	[B:5, A:2, C:1, ·]
`items[3] = items[2]; i--`	2	—	[B:5, A:2, C:1, C:1]
compare	2	`items[1].priority = 2 < 5` : true	[B:5, A:2, C:1, C:1]
`items[2] = items[1]; i--`	1	—	[B:5, A:2, A:2, C:1]
compare	1	`items[0].priority = 5 < 5` : **false**	loop ends
`items[1] = (D, 5)`	1	—	[B:5, D:5, A:2, C:1]
`size++`	—	—	`size = 4`

Table 11.6: Trace of `insert(D, 5)` on `[B:5, A:2, C:1]`. "·" marks an empty cell.

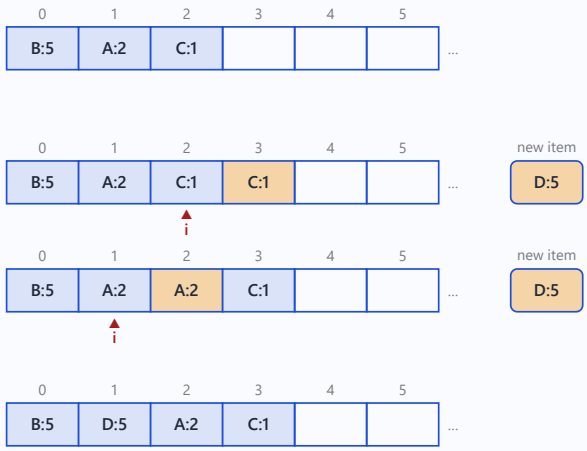


Figure 11.6: `insert(D, 5)`. (a) The initial array. (b) C has shifted right. (c) A has shifted too, `i = 1`. (d) D sits behind B; the array is still decreasing.

The key step is the comparison at `i = 1`: `5 < 5` is false, so B stays and D lands right behind it. Two `remove()` calls return B, then D, so arrival order is kept.

With `<=`, that comparison would be true, B would shift too, and D would go first: `[D:5, B:5, A:2, C:1]`. The array is still sorted, but the newcomer jumps ahead. One character turns FIFO ties into LIFO ties.

> **Common mistake**
>
> Strict (`<`) versus non-strict (`<=`) comparison in sorted insertion is easy to overlook. Both keep the array sorted and differ only in the order of equal keys, which is exactly why tests miss it.

11.2.7 Comparing the costs

Table 11.7 compares the two list implementations with the binary heap.

Operation	Unsorted list	Sorted list	Binary heap
`size`, `isEmpty`	O(1)	O(1)	O(1)
`insert`	O(1)	O(n)	O(log n)
`min`	O(n)	O(1)	O(1)
`removeMin`	O(n)	O(1)	O(log n)
Filling with *n* items	O(n)	O(n log n)*	O(n)**

Table 11.7: Operation costs of priority queue implementations.

> * By sorting the items first. ** By bottom-up heap construction.

The last row covers building a queue from *n* items given at once. An unsorted list just copies them; a sorted list must sort them, $O(n \log n)$; and a heap can be built in place in $O(n)$ by a special algorithm instead of *n* insertions.

What usually matters is the total cost of a sequence. Suppose we insert *n* items into an empty queue and then call `removeMin` *n* times.

- **Unsorted list:** $n \cdot O(1) + n \cdot O(n) = O(n^2)$. Insertions are cheap, but each removal scans what remains.
- **Sorted list:** $n \cdot O(n) + n \cdot O(1) = O(n^2)$. Now the insertions are expensive.
- **Binary heap:** $2n \cdot O(\log n) = O(n \log n)$.

For $n = 10^6$, $n^2 = 10^{12}$ while $n \log_2 n \approx 10^6 \cdot 19.93 \approx 2 \cdot 10^7$, a ratio of $n / \log_2 n \approx 50{,}000$. A job a heap finishes in one second would take a list about 50,000 seconds, roughly fourteen hours. Asymptotic analysis predicts gaps like this in advance.

11.3 Binary heaps

The list implementations are two extremes: the unsorted list keeps no order, so insertion is free; the sorted list keeps full order, so removal is free. The binary heap keeps a *partial* order, strong enough to hold the top item in a fixed place, loose enough that each operation moves only a few items.

11.3.1 Two properties

The **binary heap** is the most common way to implement a priority queue. Its items sit in the nodes of a binary tree that satisfies two conditions.

Definition 11.3

A **max-heap** is a binary tree with two properties:

1. **Heap-order property:** each node's key is greater than or equal to its children's keys.
2. **Complete binary tree property:** all levels except possibly the last are full, and the last level is filled from left to right without gaps.

With "less than or equal to" in the first property, the tree is a **min-heap**.

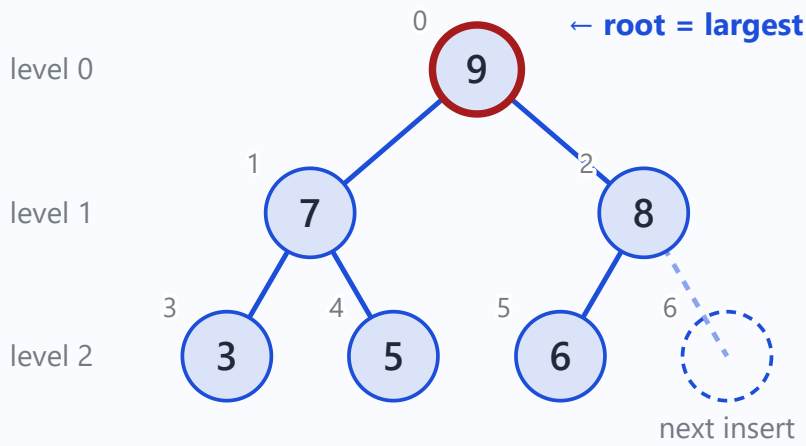


Figure 11.7: A six-node max-heap. The root, 9, is the largest key. The last level fills from the left; the next insertion goes to position 6, the right child of 8.

In Figure 11.7, the root 9 has children 7 and 8; 7 has children 3 and 5, and 8 has one child, 6. Every parent exceeds its children (9 ≥ 7, 8; 7 ≥ 3, 5; 8 ≥ 6), and the last level's only gap is at the far right.

Heap order is a local rule about parents and children, but chained down the tree it says that every node is at least as large as everything in its subtree. The root's subtree is the whole tree, so **the largest key is at the root**, and peeking at the top item takes *O(1)*.

The rule says nothing about siblings: swap 7 and 8 in Figure 11.7 and the tree is still a heap. So a heap is not sorted; apart from the root, a node's position doesn't tell you its rank.

The second property fixes the shape. A complete binary tree is the shortest binary tree with its number of nodes, so its height is $\lfloor \log_2 n \rfloor$, and it can be stored in an array with no gaps and no pointers.

Together, the two properties make heap operations fast. Insertion and removal work along *one path* between the root and a leaf, no longer than the height, so both take $O(\log n)$.

Common mistake

A heap is not a binary search tree. In a BST, left-subtree keys are smaller than the node and right-subtree keys larger, which makes searching fast. A heap has no order between left and right children, so finding a given value means checking every node: $O(n)$.

11.3.2 Min-heaps and max-heaps

Min-heaps and max-heaps differ in the direction of the inequality. In a **min-heap**, $key(parent) \le key(child)$; the smallest key is at the root, and the heap implements a min-priority queue. In a **max-heap**, $key(parent) \ge key(child)$, and the root holds the largest key.

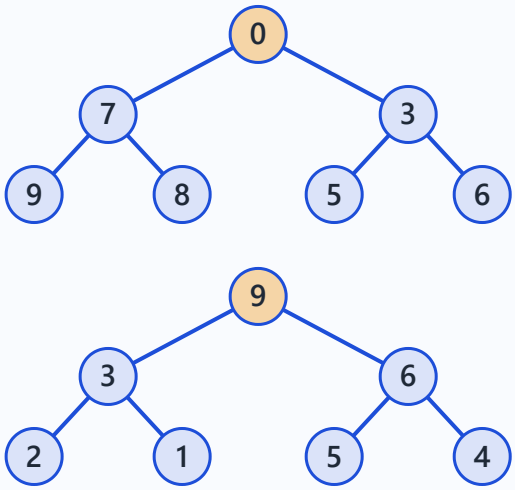


Figure 11.8: Left, a min-heap: root 0, children 7 and 3; 7 has children 9 and 8, and 3 has 5 and 6. Right, a max-heap: root 9, children 3 and 6; 3 has children 2 and 1, and 6 has 5 and 4.

In the min-heap of Figure 11.8, the siblings 7 and 3 have 7 > 3, which is fine: order holds only between parent and child. 7 even exceeds 5 and 6, the children of 3; nodes on different branches need not be related.

Geometrically, heap order says every root-to-leaf path is sorted: increasing in a min-heap, decreasing in a max-heap, as with 9 → 3 → 2, 9 → 3 → 1, 9 → 6 → 5, and 9 → 6 → 4 in the figure. A heap is a bundle of sorted paths joined at the root, and insertion and removal each repair just one of them.

11.3.3 Complete binary trees and height

A complete tree is easily confused with a perfect one. The difference is the last level, which in a complete tree need not be full, only packed *from the left without gaps*. Figure 11.9 compares three trees.

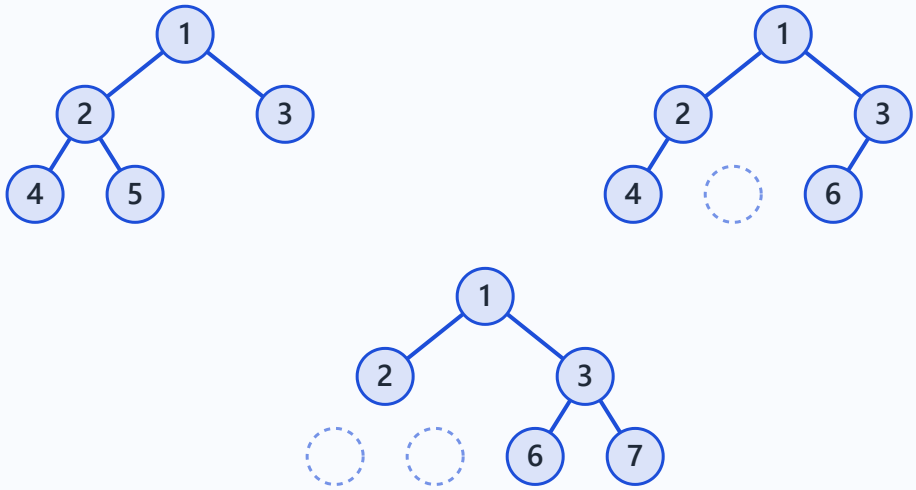


Figure 11.9: Left, complete: the last level fills from the left. Middle, not complete: there is a gap between 4 and 6. Right, not complete: the last level is not flush left.

In the left tree, root 1 has children 2 and 3, and 4 and 5 hang under 2, leaving the space under 3 empty; the tree is complete. In the middle tree, 2 lacks a right child while 3 has a left one, leaving a gap. In the right tree, both last-level nodes are under 3 and the positions under 2 are empty. Neither of these two can be a heap.

Definition 11.4

The root is at **level** 0, and each node's level is one more than its parent's. A tree's **height** is the level of its deepest leaf. Level d holds at most 2^d nodes. A **complete binary tree** of height h has levels 0, 1, ..., $h - 1$ full and the nodes of level h in the leftmost positions.

Theorem 11.1 (Height of a complete binary tree)

A complete binary tree with $n \geq 1$ nodes has height $h = \lfloor \log_2 n \rfloor$.

Proof. Levels 0 through $h - 1$ are full, holding $1 + 2 + 4 + \ldots + 2^{h-1} = 2^h - 1$ nodes, and the last level holds between 1 and 2^h. Hence $2^h \le n \le 2^{h+1} - 1 < 2^{h+1}$. Taking base-2 logarithms gives $h \le \log_2 n < h + 1$, and since h is an integer, $h = \lfloor \log_2 n \rfloor$. ■

Two checks: for $n = 7$, $\log_2 7 \approx 2.81$ and $h = 2$, since a root, two children, and four grandchildren fill three levels. For $n = 1000$, $2^9 = 512 \le 1000 < 1024 = 2^{10}$, so $h = 9$: a thousand-item heap has root-to-leaf paths of only 9 edges, and a million items need just 19. Multiplying the number of items by a thousand adds only 10 steps.

Completeness also means a new node's place is never in doubt: the leftmost free spot on the last level, or the far left of a new level if the last one is full. In the array this is exactly index `n`. Likewise, the only node that can be removed without spoiling the shape is the rightmost on the last level, the array's last element.

11.3.4 Heap or not?

To decide whether a tree is a heap, check the two properties separately: is the shape complete, and does every parent–child pair satisfy the inequality in the same direction?

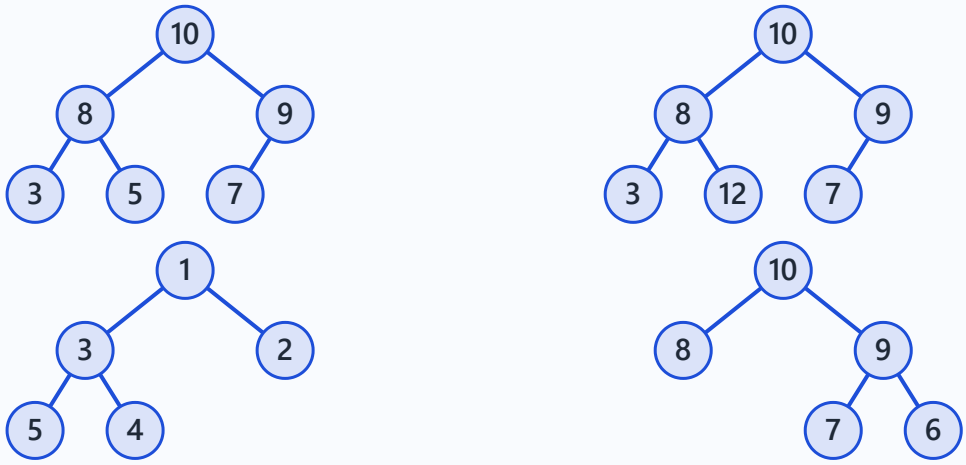


Figure 11.10: Four trees: (a) top left, (b) top right, (c) bottom left, (d) bottom right.

Example 11.3

Is each tree in Figure 11.10 a max-heap, a min-heap, or not a heap?

1. **(a)** Root 10; children 8 and 9; 8 has children 3 and 5, and 9 has a left child, 7. The tree is complete: 3, 5, 7 fill the last level from the left. All pairs go one way (10 ≥ 8, 10 ≥ 9, 8 ≥ 3, 8 ≥ 5, 9 ≥ 7): **max-heap**.
2. **(b)** Same shape, but 8's right child is 12 > 8, breaking max-heap order, and 10 ≥ 8 rules out a min-heap: **not a heap**.

3. **(c)** Root 1; children 3 and 2; 3 has children 5 and 4. Complete, and $1 \le 3$, $1 \le 2$, $3 \le 5$, $3 \le 4$: **min-heap**.
4. **(d)** Root 10; children 8 and 9; 8 has none, 9 has 7 and 6. Every parent exceeds its children, but the last level is not flush left, so the tree is not complete: **not a heap**.

Tree (d) shows that heap order alone is not enough. A bad shape leaves gaps in the array: (d) must be stored as `[10, 8, 9, _, _, 7, 6]`. The index formulas and the rule "a new item goes at index `n`" both assume there are no gaps.

11.3.5 Code trace: isHeap()

Checking whether an array is a max-heap is easy. An array has no gaps, so the tree it represents is automatically complete, and only heap order needs checking. The node at index `i` has its parent at `(i - 1) / 2` (Definition 11.5).

```
static boolean isHeap(int[] a) {  // max-heap?
  for (int i = 1; i < a.length; i++) {
    int p = (i - 1) / 2;          // parent
    if (a[p] < a[i]) return false; // order violated
  }
  return true;
}
```

The loop visits each non-root node, `i = 1` to `n - 1`, and compares it with its parent. A parent smaller than its child violates the order, and the method returns `false` at once; if the loop finishes, the array is a max-heap.

Every node has exactly one parent, so each pair is checked exactly once. Looping over children is also simpler than looping over parents, since there is no need to ask whether a right child exists.

Here is the trace on `a = [10, 8, 9, 3, 5, 7, 4, 6]`.

`i`	`p = (i - 1) / 2`	Comparison	Result
1	0	`a[0] = 10 ≥ a[1] = 8`	continue
2	0	`a[0] = 10 ≥ a[2] = 9`	continue
3	1	`a[1] = 8 ≥ a[3] = 3`	continue
4	1	`a[1] = 8 ≥ a[4] = 5`	continue
5	2	`a[2] = 9 ≥ a[5] = 7`	continue
6	2	`a[2] = 9 ≥ a[6] = 4`	continue
7	3	`a[3] = 3 < a[7] = 6`	`return false`

Table 11.8: Trace of `isHeap([10, 8, 9, 3, 5, 7, 4, 6])`.



Figure 11.11: At `i = 7` the violation is found: 6 is larger than its parent, 3.

The first seven items, `[10, 8, 9, 3, 5, 7, 4]`, form a valid max-heap, and on them the method returns `true` in Java. The appended 6 exceeds its parent 3. This is why insertion needs a repair step: writing the new item at the end keeps the shape but not necessarily the order.

In the worst case, when the array is a heap, the loop makes $n - 1$ comparisons: $O(n)$ time and $O(1)$ extra space. No check can be faster, since any skipped pair might hide a violation.

For empty and one-element arrays the loop never runs and the method returns `true`, correctly: with no parent–child pairs, heap order holds trivially. For a min-heap, reverse the test: `if (a[p] > a[i]) return false;`.

11.3.6 The array representation

Trees are usually built from node objects and references, but a heap needs neither. Its nodes are written into an array in level order, level by level and left to right: the root in `heap[0]`, its children in `heap[1]` and `heap[2]`, the grandchildren in `heap[3]` through `heap[6]`, and so on.

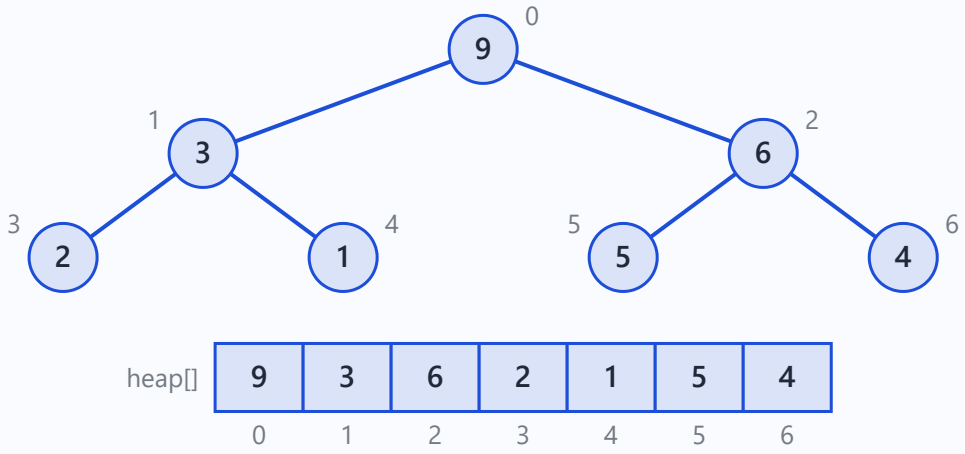


Figure 11.12: A max-heap and its array. The small numbers beside the nodes are array indices. Array: [9, 3, 6, 2, 1, 5, 4].

Because the tree is complete, level order leaves no gaps (Figure 11.12): `n` items fill exactly `heap[0]` through `heap[n - 1]`, and parent and child links are computed from the indices rather than stored.

This has three practical advantages. It saves memory, since no child references are stored. It is fast, since contiguous cells reach the processor's cache in blocks, and walking an array beats chasing scattered nodes. And it is simple: the last item and the first free position are both given by one counter, `n`.

These notes index from 0, with the root at `heap[0]`. Some books start at 1; the two are equivalent, with slightly different formulas.

11.3.7 Parent and child indices

In level order, a node's children and parent follow from simple formulas.

Definition 11.5

In the 0-based representation, the node at index `i` has

- left child $left(i) = 2i + 1$,
- right child $right(i) = 2i + 2$,
- parent $parent(i) = \lfloor (i - 1) / 2 \rfloor$ (for $i \geq 1$).

A child index of n or more means there is no such child; in particular, `i` is a leaf if $2i + 1 \geq n$.

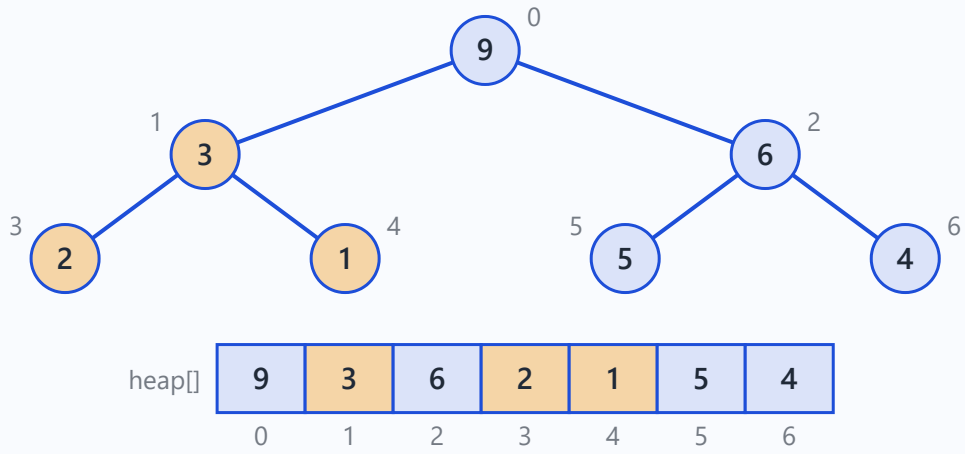


Figure 11.13: The 3 at index 1 has its children, 2 and 1, at indices 3 and 4.

To see why, note that level d starts at index $2^d - 1$ (0, 1, 3, 7, ...). Since each node takes two children in turn, the children of a level's jth node (counting from 0) are the $2j$th and $(2j + 1)$th nodes of the next level. In indices, the node is $i = 2^d - 1 + j$ and its left child is $2^{d+1} - 1 + 2j = 2(2^d - 1 + j) + 1 = 2i + 1$; the right child is the next cell. The parent formula inverts this: subtracting 1 from $2i + 1$ or $2i + 2$ and halving, rounded down, gives i.

Example 11.4

Check the formulas on `[9, 3, 6, 2, 1, 5, 4]` (Figure 11.13).

1. `i = 1` (3): children at $2 \cdot 1 + 1 = 3$ and $2 \cdot 1 + 2 = 4$, holding 2 and 1.
2. `i = 2` (6): children at 5 and 6, holding 5 and 4.
3. `i = 6` (4): parent $\lfloor (6 - 1)/2 \rfloor = \lfloor 2.5 \rfloor = 2$, the 6; for `i = 5`, also $\lfloor 4/2 \rfloor = 2$. Siblings share a parent.
4. `i = 4` (1): parent $\lfloor 3/2 \rfloor = 1$, the 3. Its first child would be $2 \cdot 4 + 1 = 9 \geq n = 7$, so it is a leaf.
5. A larger case: index 13 has parent $\lfloor 12/2 \rfloor = 6$ and children 27 and 28.

Java's integer division rounds positive numbers down, so the parent is simply `(i - 1) / 2` in code, and the child formulas need only addition and multiplication. Moving to a neighbor in a heap costs no more than following a reference.

1-based representation. Some books leave `heap[0]` empty and put the root at `heap[1]`; then node k has children $2k$ and $2k + 1$ and parent $\lfloor k/2 \rfloor$. The formulas are shorter at the cost of one cell. Either works, as long as you use one consistently.

Common mistake

Don't mix the 0-based and 1-based formulas. With the root at `heap[0]`, writing `2 * i` for the left child makes the root its own child ($2 \cdot 0 = 0$). 0-based: $2i + 1$, $2i + 2$, $(i - 1)/2$; 1-based: $2i$, $2i + 1$, $i/2$.

11.4 Insertion and removal

Both basic heap operations follow one pattern. First they preserve the *shape*, putting the item into, or taking it from, the one place where completeness survives. This may break heap order at one spot, and the second step repairs it by moving the item up or down a single path: **up-heap** or **down-heap**. The code in this section is for a max-heap.

11.4.1 Insertion: up-heap

Insertion into a max-heap takes three steps:

1. **Preserve the shape.** Write the item into the first free position, `heap[n]`, and increment `n`.
2. **Find the defect.** Order can now fail only between the new item and its parent. If the item is larger, swap them.
3. **Repeat** with the new parent, until the item reaches the root or its parent is at least as large.

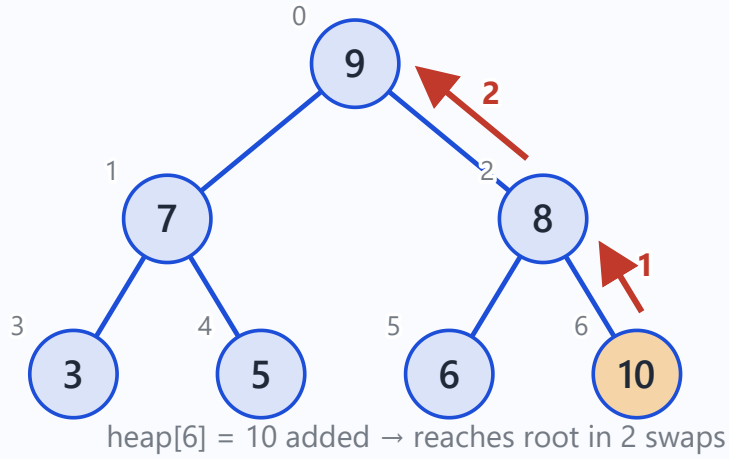


Figure 11.14: Inserting 10 into [9, 7, 8, 3, 5, 6]: 10 goes to index 6, then swaps with 8 and with 9, reaching the root in two swaps.

In Figure 11.14, 10 enters at index 6 as the right child of 8. Since 8 < 10, they swap and 10 moves to index 2; since the root 9 < 10, they swap again. 10 is now the root, and the result is `[10, 7, 9, 3, 5, 6, 8]`.

The operation goes by many names (up-heap, bubble-up, sift-up, swim). Our code calls it `swim`: a large key rises toward the root like a buoyant object swimming to the surface.

Theorem 11.2 (Correctness and cost of up-heap)

Adding an item at the end of a valid max-heap and applying up-heap yields a valid max-heap, after at most $\lfloor \log_2 n \rfloor$ swaps and in $O(\log n)$ time.

Proof. Call the rising item x. At every step, heap order holds everywhere except possibly between x and its parent. Initially this is true, since only a new leaf was added to a heap. Suppose the parent p is smaller than x, and they swap. p moves down and becomes the parent of x's former children, which were p's grandchildren and so no larger than p; p's other child was no larger either. So p dominates everything below it. In its new place, x exceeds p and hence p's other child. The one possible violation has moved up a level, and it disappears when x reaches the root or a parent $\geq x$. Each swap raises x one level, so there are at most $\lfloor \log_2 n \rfloor$ swaps. ■

In short, the parent that moves down was already at least as large as its other child, so fixing the defect above never creates one below.

11.4.2 Code: insert() and swim()

The max-heap class keeps an `int` array and a counter:

```
public class MaxHeap {
  private int[] heap = new int[4];
  private int n = 0;                // number of items

public void insert(int x) {
  if (n == heap.length) resize(2 * heap.length);
  heap[n] = x;  n++;              // first free position
  swim(n - 1);
}
private void swim(int i) {        // up-heap
  while (i > 0) {
    int parent = (i - 1) / 2;
    if (heap[parent] >= heap[i]) return;
    swap(i, parent);  i = parent;
  }
}
private void swap(int a, int b) {
  int t = heap[a];  heap[a] = heap[b];  heap[b] = t;
}
}  // MaxHeap
```

The capacity starts at 4. `insert` doubles it with `resize` if the array is full (`n == heap.length`), writes `x` into `heap[n]`, increments `n`, and calls `swim` on the new index, `n - 1`.

`swim(i)` loops while `i` is not the root, computing the parent's index as `(i - 1) / 2`. If the parent is at least as large, the order holds and the method returns; otherwise `swap`, a three-assignment helper with a temporary variable, exchanges the two cells, and `i` moves up to the parent.

The loop exits in two ways: through the `while` test when `i = 0` (the root has no parent), or through the inner `return` when the parent is ≥ the item, since the path above is already in order. Using `>=` avoids pointless swaps of equal keys.

Cost: `swim` makes at most $\lfloor \log_2 n \rfloor$ constant-time passes, $O(\log n)$. Apart from `resize`, `insert` does constant work, and the rare resizes cost amortized $O(1)$ per insertion. Insertion is thus amortized $O(\log n)$.

11.4.3 Code trace: insert(10), insert(8)

The heap starts as `[9, 3, 6, 2, 1, 5, 4]`, with `n = 7` and capacity 16; the largest item, 9, is at the root. We call `insert(10)`, then `insert(8)`.

Step	`i`	`parent`	Comparison	Array
`insert(10)`: `heap[7] = 10`	7	—	—	[9, 3, 6, 2, 1, 5, 4, 10], `n = 8`
pass 1	7	3	`heap[3] = 2 < 10`: swap	[9, 3, 6, 10, 1, 5, 4, 2]
pass 2	3	1	`heap[1] = 3 < 10`: swap	[9, 10, 6, 3, 1, 5, 4, 2]
pass 3	1	0	`heap[0] = 9 < 10`: swap	[10, 9, 6, 3, 1, 5, 4, 2]

Step	`i`	`parent`	Comparison	Array
loop ends	0	—	`i > 0` false	10 at the root
`insert(8)` : `heap[8] = 8`	8	—	—	[10, 9, 6, 3, 1, 5, 4, 2, 8], `n = 9`
pass 1	8	3	`heap[3] = 3 < 8` : swap	[10, 9, 6, 8, 1, 5, 4, 2, 3]
pass 2	3	1	`heap[1] = 9 ≥ 8` : `return`	[10, 9, 6, 8, 1, 5, 4, 2, 3]

Table 11.9: Trace of `insert(10)` and `insert(8)` on `[9, 3, 6, 2, 1, 5, 4]`.

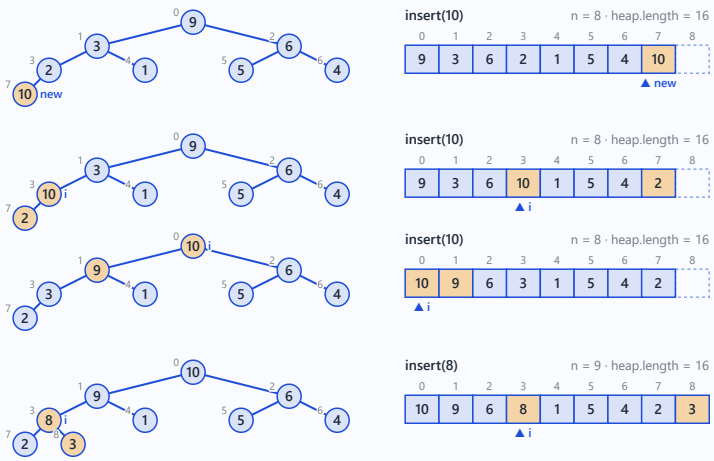


Figure 11.15: (a) 10 is written to `heap[7]`. (b) After one swap, 10 is at index 3. (c) Three swaps bring it to the root. (d) `insert(8)` takes one swap: 8 is at index 3, below the larger 9.

`insert(10)` is the worst case: the new item is the largest and climbs from leaf to root in 3 swaps, the height $\lfloor \log_2 8 \rfloor = 3$. `insert(8)` stops halfway: after swapping with 3, it meets $9 \geq 8$ and returns. That makes 4 swaps in all.

Note that the first parent compared in `insert(8)` is the 3 that `insert(10)` pushed from index 1 down to index 3. Each up-heap pushes the items on its path down a level, and later insertions work on the rearranged heap.

11.4.4 Filling an empty heap

To build a heap from scratch, insert 4, 5, 2, 6, 1, 3 into an empty max-heap of capacity 4 (Table 11.10).

Call	Written to	Up-heap steps	Resulting array	Capacity
`insert(4)`	`heap[0]`	`i = 0` : root, loop not entered	[4]	4
`insert(5)`	`heap[1]`	parent 0: 4 < 5, swap; `i = 0`	[5, 4]	4
`insert(2)`	`heap[2]`	parent 0: 5 ≥ 2, `return`	[5, 4, 2]	4
`insert(6)`	`heap[3]`	parent 1: 4 < 6, swap; parent 0: 5 < 6, swap	[6, 5, 2, 4]	4
`insert(1)`	`heap[4]`	first `resize(8)` ; parent 1: 5 ≥ 1, `return`	[6, 5, 2, 4, 1]	8

Call	Written to	Up-heap steps	Resulting array	Capacity
`insert(3)`	`heap[5]`	parent 2: 2 < 3, swap; parent 0: 6 ≥ 3, `return`	[6, 5, 3, 4, 1, 2]	8

Table 11.10: Successive insertions into an empty heap (capacity 4).

The first item simply becomes the root. `insert(6)` brings the largest item so far, which climbs to the root. At `insert(1)` the array is full (`n = 4 = heap.length`), so the capacity grows to 8 first; 1 is smaller than its parent 5 and stays put. Finally, 3 swaps with its parent 2 and settles under 6.

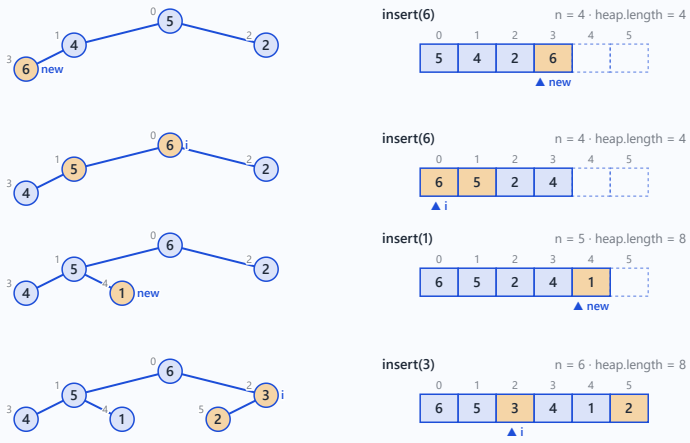


Figure 11.16: (a) `insert(6)`: 6 at index 3. (b) Two swaps bring it to the root: [6, 5, 2, 4]. (c) After resizing, 1 goes to index 4 and stays. (d) `insert(3)`: 3 swaps with its parent 2: [6, 5, 3, 4, 1, 2].

The result, `[6, 5, 3, 4, 1, 2]`, is not sorted but is a valid max-heap: 6 ≥ 5, 3; 5 ≥ 4, 1; 3 ≥ 2. Another insertion order could give another heap: the shape is fixed, but where the items land depends on the order of insertion.

11.4.5 Growing the capacity

A Java array can't change length, so growing the capacity means allocating a larger array and copying:

```
private void resize(int capacity) {
  int[] newArr = new int[capacity];
  for (int k = 0; k < n; k++)
    newArr[k] = heap[k];        // copy
  heap = newArr;                // old array becomes garbage
}
```

`resize(capacity)` allocates the new array, which Java fills with zeros, copies the first `n` items to the same indices, and points `heap` at the new array. The old array is no longer referenced, and the garbage collector reclaims it.

Step	`k`	`newArr`
`new int[8]`	—	[0, 0, 0, 0, 0, 0, 0, 0]
`newArr[0] = heap[0]`	0	[6, 0, 0, 0, 0, 0, 0, 0]
`newArr[1] = heap[1]`	1	[6, 5, 0, 0, 0, 0, 0, 0]
`newArr[2] = heap[2]`	2	[6, 5, 2, 0, 0, 0, 0, 0]
`newArr[3] = heap[3]`	3	[6, 5, 2, 4, 0, 0, 0, 0]
`k = 4 = n`: loop ends	4	4 items copied
`heap = newArr`	—	same tree, capacity 8

Table 11.11: Trace of `resize(8)` on `[6, 5, 2, 4]`.

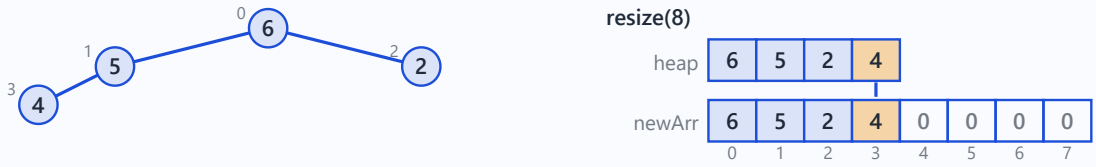


Figure 11.17: The copy is nearly done: the old array's four items occupy the first four cells of the new one; the rest are 0.

One resize costs *O(n)*, but resizes are rare. With doubling, they happen at 4, 8, 16, ... items, so n insertions copy $4 + 8 + 16 + \ldots$ items, a sum whose last term is below n. Since each term doubles the one before, the sum is less than twice the last term, under $2n$ in all: a constant per insertion, so resizing is **amortized** *O(1)*.

Growing by a fixed amount, say 4 cells, would be a bad choice: a resize every 4 insertions copies $4 + 8 + 12 + \ldots \approx n^2/8$ items, *O(n)* per insertion.

Removal can shrink the array symmetrically, halving it when it falls to a quarter full. Why a quarter rather than a half? At a half, alternating insertions and removals at the threshold would resize every time. Shrinking is optional and is omitted here.

> **Common mistake**
>
> Growing with `heap = new int[2 * heap.length]` and forgetting to copy loses the whole heap, because the new array is all zeros. Always allocate, copy, then switch the reference; `Arrays.copyOf(heap, newLength)` does all three in one line.

11.4.6 Removing the maximum: down-heap

The largest item of a max-heap is at the root. Removing it also has two phases.

1. **Save the root.** It is the return value.

2. **Preserve the shape.** Only the last node can leave without spoiling completeness, so move `heap[n - 1]` to the root and decrement `n`.
3. **Repair.** The new root is probably small and may be smaller than its children. Swap it with the *larger* child.
4. **Repeat** until it reaches a leaf or is at least as large as both children.

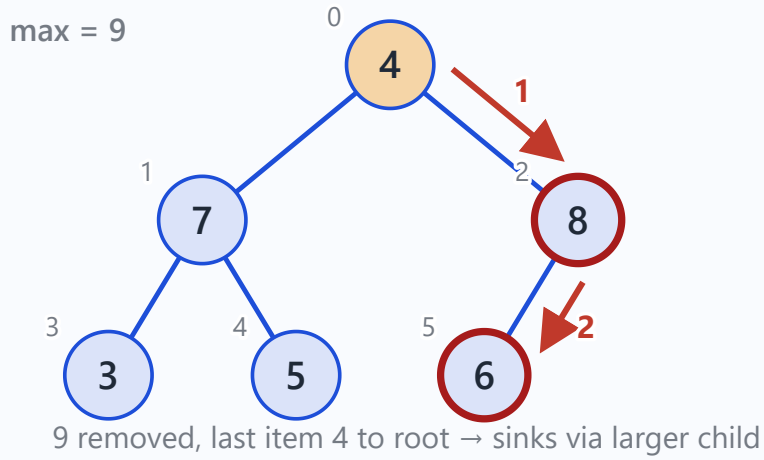


Figure 11.18: After 9 is removed from [9, 7, 8, 3, 5, 6, 4], the last item, 4, moves to the root and swaps first with its larger child 8, then with 6.

In Figure 11.18, 4 replaces the removed 9 at the root. Its children are 7 and 8, and it swaps with the larger, 8. At index 2 its only child is 6, and 4 < 6, so it swaps again and reaches a leaf. The result is `[8, 7, 6, 3, 5, 4]`.

Why the larger child? The child that moves up becomes the parent of its sibling. If it is the larger one, order holds at the new root; if the smaller moved up, it would sit above a larger child, and the defect would merely move. In a min-heap the same argument picks the smaller child.

Cost: two comparisons per level, one to pick the larger child and one against the item, over at most $h = \lfloor \log_2 n \rfloor$ levels, for $O(\log n)$ in total.

11.4.7 Code: removeMax() and sink()

```
public int removeMax() {        // requires n > 0
  int max = heap[0];
  swap(0, n - 1);  n--;         // last item to the root
  sink(0);
  return max;
}
private void sink(int i) {        // down-heap
  while (2 * i + 1 < n) {       // left child exists?
    int j = 2 * i + 1;            // j: larger child
    if (j + 1 < n && heap[j + 1] > heap[j]) j++;
    if (heap[i] >= heap[j]) return;
    swap(i, j);  i = j;           // down one level
  }
}
```

`removeMax` saves the root in `max`, swaps it with the last item, and decrements `n`. The swap also leaves the old root at the end of the array, outside the heap, which heap sort exploits: repeated removals collect the largest items at the end. Then `sink(0)` runs and `max` is returned. A real class would also check `if (n == 0)` and throw an exception, since `heap[0]` means nothing in an empty heap.

`sink(i)`, named for an object sinking in water, is down-heap. The loop test `2 * i + 1 < n` asks whether `i` has a left child; in a complete tree a node without one has no right child either, so the loop ends at a leaf. Inside, `j` starts at the left child and moves to the right one if it exists and is larger. If the item is at least as large as child `j`, the method returns; otherwise they swap and the item drops a level.

The test `j + 1 < n` is critical. After `removeMax`, the old root still sits at index `n`; without the test, `heap[j + 1]` could read it, take it for the larger child, and move it back into the heap (Example 11.5).

11.4.8 Code trace: removeMax()

Call `removeMax()` on `[9, 3, 6, 2, 1, 5, 4]` (`n = 7`).

Step	`i`	Children → `j`	Decision	Array
`max = heap[0]`	—	—	`max = 9`	[9, 3, 6, 2, 1, 5, 4]
`swap(0, 6); n--`	—	—	`n = 6`	[4, 3, 6, 2, 1, 5 | 9]
pass 1	0	1 (3), 2 (6) → `j = 2`	4 < 6: swap	[6, 3, 4, 2, 1, 5 | 9]
pass 2	2	only 5 (5) → `j = 5`	4 < 5: swap	[6, 3, 5, 2, 1, 4 | 9]
loop ends	5	`2 · 5 + 1 = 11 ≥ 6`	leaf	[6, 3, 5, 2, 1, 4 | 9]

Step	`i`	Children → `j`	Decision	Array
`return max`	—	—	returns 9	—

Table 11.12: Trace of `removeMax()` on `[9, 3, 6, 2, 1, 5, 4]`. The item to the right of the vertical bar no longer belongs to the heap.

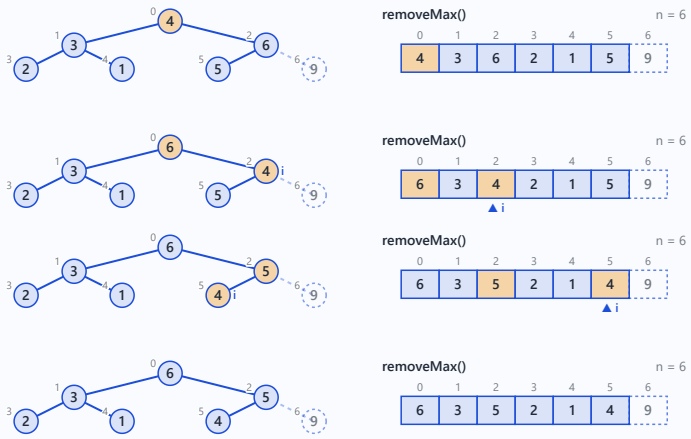


Figure 11.19: (a) Root and last item swapped; 9 is outside the heap (faded). (b) 4 swapped with its larger child, 6. (c) 4 swapped with its only child, 5. (d) Result: [6, 3, 5, 2, 1, 4].

In pass 1 the root's larger child is 6. In pass 2, the 4 at index 2 has only a left child, `j = 5`; the right candidate, 6, is not less than `n = 6`. `heap[6]` still holds 9, but the `j + 1 < n` test keeps it from being read. In all, 3 comparisons and 2 swaps; with height $\lfloor \log_2 6 \rfloor = 2$, no more swaps are possible.

11.4.9 Code trace: two removeMax() calls in a row

Next, remove the maximum twice from `[10, 9, 6, 8, 1, 5, 4, 2, 3]`, the heap left by the up-heap trace.

Step	`i`	Children → `j`	Decision	Heap part
#1: root ↔ last, `n = 8`	—	—	`max = 10`	[3, 9, 6, 8, 1, 5, 4, 2]
pass 1	0	9, 6 → `j = 1`	3 < 9: swap	[9, 3, 6, 8, 1, 5, 4, 2]
pass 2	1	8, 1 → `j = 3`	3 < 8: swap	[9, 8, 6, 3, 1, 5, 4, 2]
pass 3	3	only 2 → `j = 7`	3 ≥ 2: `return`	[9, 8, 6, 3, 1, 5, 4, 2]
#2: root ↔ last, `n = 7`	—	—	`max = 9`	[2, 8, 6, 3, 1, 5, 4]
pass 1	0	8, 6 → `j = 1`	2 < 8: swap	[8, 2, 6, 3, 1, 5, 4]
pass 2	1	3, 1 → `j = 3`	2 < 3: swap	[8, 3, 6, 2, 1, 5, 4]
loop ends	3	`2 · 3 + 1 = 7 ≥ 7`	leaf	[8, 3, 6, 2, 1, 5, 4]

Table 11.13: Two successive `removeMax()` calls on `[10, 9, 6, 8, 1, 5, 4, 2, 3]`.

The removals return 10 and 9. The first sinking item, 3, stops above a leaf: at index 3 its only child is 2. The second, 2, sinks all the way. The last two cells now hold 9 and 10, `[8, 3, 6, 2, 1, 5, 4 | 9, 10]`: removed items pile up at the end, decreasing from right to left.

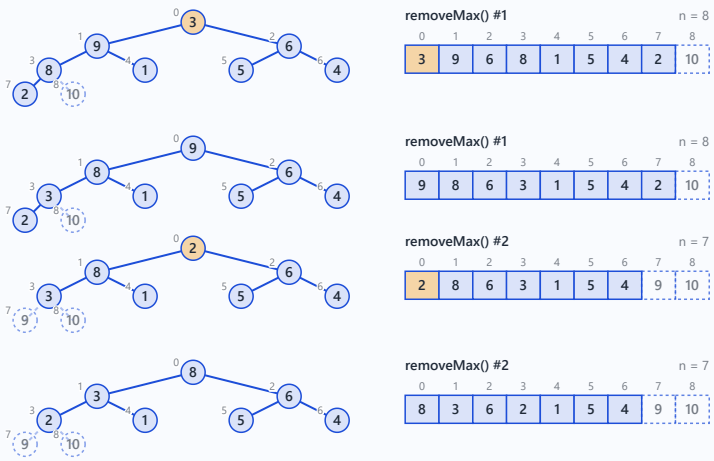


Figure 11.20: (a) First removal: 3 at the root, 10 outside the heap. (b) Done: [9, 8, 6, 3, 1, 5, 4, 2]. (c) Second removal: 2 at the root. (d) Done; 9 and 10 sit at the end of the array.

Example 11.5

Suppose `sink` omits the `j + 1 < n` test: `if (heap[j + 1] > heap[j]) j++;`. What does `removeMax()` do on `[9, 8, 3, 5, 4]` (`n = 5`)?

1. `max = 9`; 9 and 4 swap, `n = 4`: `[4, 8, 3, 5 | 9]`.
2. `i = 0`: children 8 and 3, `j = 1`; 4 < 8, swap: `[8, 4, 3, 5 | 9]`.
3. `i = 1`: the left child, 5, is at index 3. Since `heap[4] = 9 > 5`, the buggy code picks `j = 4`, although index 4 is outside the heap and holds the deleted 9. 4 < 9, swap: `[8, 9, 3, 5 | 4]`.
4. The heap is now `[8, 9, 3, 5]`: the deleted 9 is back, 4 is lost, and 9 exceeds its parent 8. Correct code gives `[8, 5, 3, 4]`. If `sink` runs without decrementing `n` on a full array (in `buildHeap`, say), the same bug throws `ArrayIndexOutOfBoundsException`.

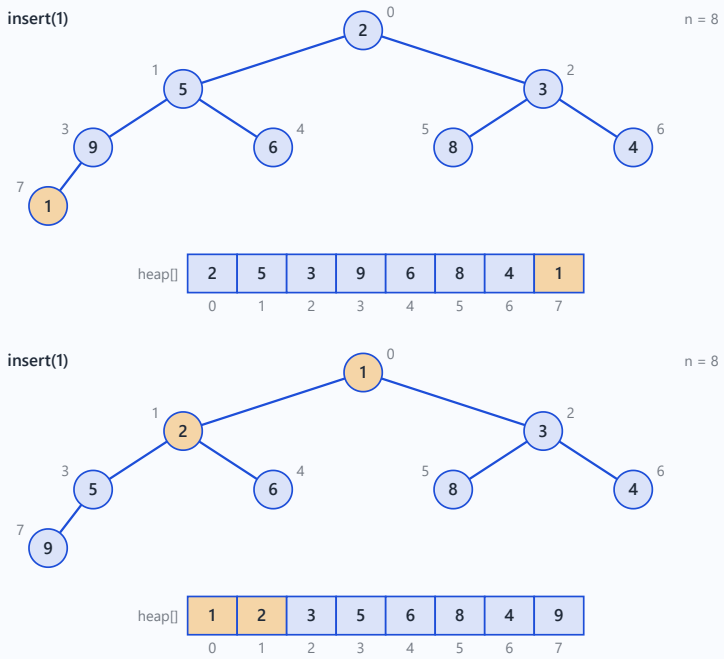
11.4.10 insert and removeMin on a min-heap

A min-heap works the same way with the comparisons reversed: up-heap moves an item up while it is *smaller* than its parent, and down-heap swaps it with the *smaller* child. Start from the min-heap `[2, 5, 3, 9, 6, 8, 4]`.

Step	Decision	Array
`insert(1)`: write at index 7	parent *(7 − 1)/2 = 3* → 9	[2, 5, 3, 9, 6, 8, 4, 1]
up-heap	1 < 9: swap (7 ↔ 3)	[2, 5, 3, 1, 6, 8, 4, 9]

Step	Decision	Array
up-heap	1 < 5: swap (3 ↔ 1)	[2, 1, 3, 5, 6, 8, 4, 9]
up-heap	1 < 2: swap (1 ↔ 0); root, stop	[1, 2, 3, 5, 6, 8, 4, 9]
`removeMin()`	`min = 1`; root ↔ last (0 ↔ 7), `n = 7`	[9, 2, 3, 5, 6, 8, 4 | 1]
down-heap	children 2, 3 → smaller 2; 9 > 2: swap (0 ↔ 1)	[2, 9, 3, 5, 6, 8, 4 | 1]
down-heap	children 5, 6 → smaller 5; 9 > 5: swap (1 ↔ 3)	[2, 5, 3, 9, 6, 8, 4 | 1]
end	index 3 is a leaf ($2 \cdot 3 + 1 = 7 \geq 7$)	returns 1

Table 11.14: `insert(1)` and `removeMin()` on the min-heap `[2, 5, 3, 9, 6, 8, 4]`.



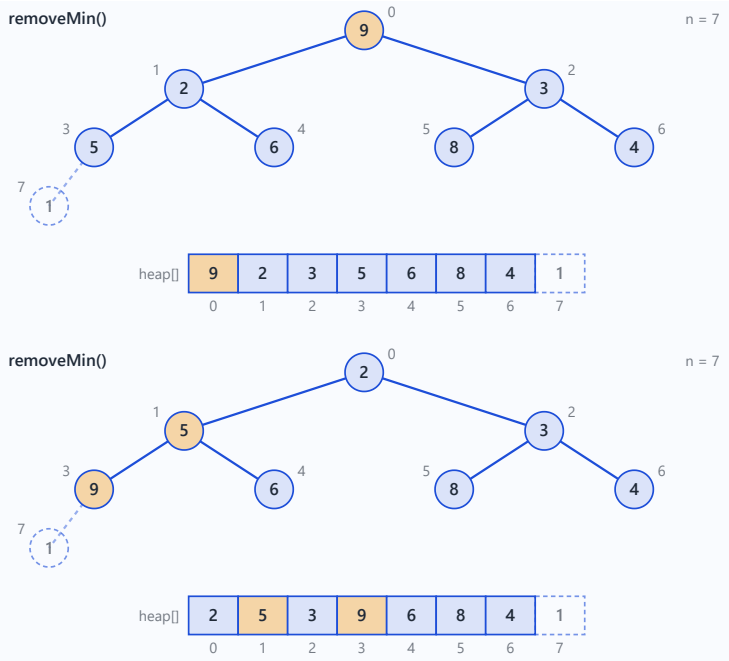


Figure 11.21: In a min-heap, (a) 1 is inserted at index 7, (b) three swaps bring it to the root, (c) `removeMin` moves 9 to the root, (d) two swaps sink 9 back.

Here the insertion and removal cancel out exactly, but that is a coincidence: the inserted item was the smallest and pushed its whole path down a level, and the 9 then sank back along the same path. In general, an insertion followed by a removal does not restore the original heap.

To turn max-heap code into min-heap code, reverse three comparisons: `heap[parent] <= heap[i]` instead of `>=` in `swim`, and in `sink` the smaller child instead of the larger, with `heap[i] <= heap[j]` as the stopping test. Alternatively, store negated keys in a max-heap: the largest $-x$ is the smallest x.

11.4.11 The cost of heap operations

Operation	Cost	Why?
`size`, `isEmpty`	O(1)	The counter `n` is maintained.
`max` / `min` (peek)	O(1)	Always at `heap[0]`.
`insert`	O(log n)*	Up-heap along one leaf-to-root path.
`removeMax` (removeMin)	O(log n)*	Down-heap along one root-to-leaf path.
Searching for an arbitrary item	O(n)	A heap is not sorted, so it doesn't support search.

Table 11.15: Costs of binary heap operations.

> * Amortized, including growing and shrinking the array.

The height formula makes this concrete: a heap of $n = 10^6$ items has height $\lfloor \log_2 10^6 \rfloor = 19$, so an insertion or removal makes at most 19 swaps, where a sorted array might shift half a million items.

Memory is $O(n)$, and doubling leaves at most half the array empty. A linked tree would instead store two child references, and often a parent reference, in every node.

The last row of the table is a reminder: a heap finds only the top item quickly. “Is 7 in the queue?” requires a full scan, and changing or removing a particular item requires finding it first. The adaptable priority queue addresses this.

11.4.12 Adaptable priority queues

Sometimes an item's priority changes while it waits. In Dijkstra's algorithm, a shorter path to a node lowers its distance, which is its key; in an emergency room, a waiting patient gets worse. And sometimes an item leaves early: a patient gives up, a process is canceled.

Definition 11.6

An **adaptable priority queue** adds two operations to the basic ones:

- `replaceKey(e, k)` : sets the key of item `e` to `k` ;
- `remove(e)` : removes item `e` and returns it.

Here `e` is the item object returned to the user on insertion.

The hard part is knowing where `e` is in the heap array, since searching an unsorted heap takes $O(n)$. Goodrich and his coauthors use a **location-aware entry**: each item object stores its current index alongside its key and value, and every swap updates the indices of both items.

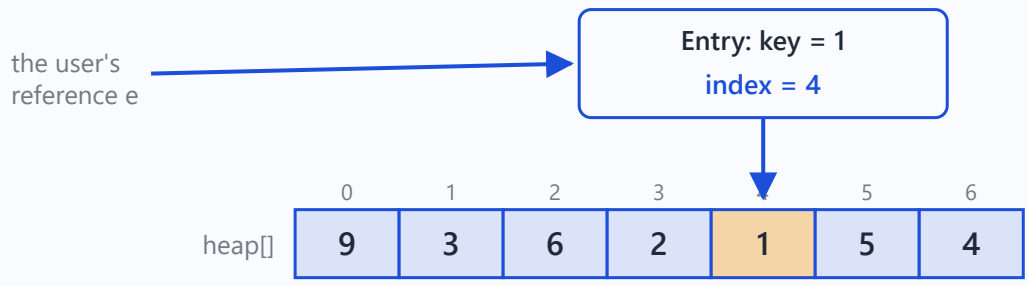


Figure 11.22: A location-aware entry. The outside reference `e` points to the entry with key 1, which knows its index (4) in the heap array.

With the location known, both operations use familiar tools:

- `replaceKey(e, k)`. Change the key of `heap[e.index]`. In a max-heap, a larger key can conflict only with the parent, so apply up-heap; a smaller key only with the children, so apply down-heap.
- `remove(e)`. Swap the item with the last one and decrement `n`. The replacement may be larger than its parent or smaller than its children, so apply up-heap or down-heap as needed.

Without the location field, each operation would start with an *O(n)* search; with it, the item is reached in *O(1)* and the operation costs *O(log n)*. The price is updating two indices per swap, a constant overhead.

Common mistake

In `remove(e)`, don't apply only down-heap to the replacement. The last item may come from another branch and be larger than the removed item's parent. Removing the 1 at index 3 of `[9, 3, 8, 1, 2, 7, 6]`, for example, brings in 6, which exceeds its new parent 3: up-heap is needed.

11.4.13 Code trace: increasing and decreasing a key

Our simple max-heap stores integers rather than item objects, so an item is identified by its index: `changeKey(i, key)` sets `heap[i]` to `key` and repairs the order.

```
public void changeKey(int i, int key) {
  int old = heap[i];
  heap[i] = key;
  if (key > old) swim(i);      // increased: up
  else sink(i);                // decreased: down
}
private void swim(int i) {       // up-heap
  while (i > 0) {
    int parent = (i - 1) / 2;
    if (heap[parent] >= heap[i]) return;
    swap(i, parent);  i = parent;
  }
}
```

The method saves the old value, writes the new one, and picks a direction: `swim(i)` if the key grew, `sink(i)` otherwise. For an unchanged key, `sink` returns at once, since the item already dominates its children. The shape never changes; only values along one path move.

Increasing a key. Call `changeKey(4, 8)` on `[9, 3, 6, 2, 1, 5, 4]`.

Step	`i`	Decision	Array
`old = heap[4]`	4	`old = 1`	[9, 3, 6, 2, 1, 5, 4]
`heap[4] = 8`	4	8 > 1: increased, `swim(4)`	[9, 3, 6, 2, 8, 5, 4]

Step	`i`	Decision	Array
pass 1	4	parent 1: 3 < 8, swap	[9, 8, 6, 2, 3, 5, 4]
pass 2	1	parent 0: 9 ≥ 8, `return`	[9, 8, 6, 2, 3, 5, 4]

Table 11.16: `changeKey(4, 8)` on `[9, 3, 6, 2, 1, 5, 4]`.

8 rose one level, stopping below the root, and only the path 4 → 1 → 0 was examined. As in the proof, the 3 that moved down lands in a leaf, and at index 1 it already exceeded its other child, 2.

Decreasing a key. Now decrease the root of the updated heap `[9, 8, 6, 2, 3, 5, 4]`: `changeKey(0, 1)`.

Step	`i`	Children → `j`	Decision	Array
`old = heap[0]`; `heap[0] = 1`	0	—	1 < 9: decreased, `sink(0)`	[1, 8, 6, 2, 3, 5, 4]
pass 1	0	8, 6 → `j = 1`	1 < 8: swap	[8, 1, 6, 2, 3, 5, 4]
pass 2	1	2, 3 → `j = 4`	1 < 3: swap	[8, 3, 6, 2, 1, 5, 4]
loop ends	4	`2 · 4 + 1 = 9 ≥ 7`	leaf	[8, 3, 6, 2, 1, 5, 4]

Table 11.17: `changeKey(0, 1)` on `[9, 8, 6, 2, 3, 5, 4]`.

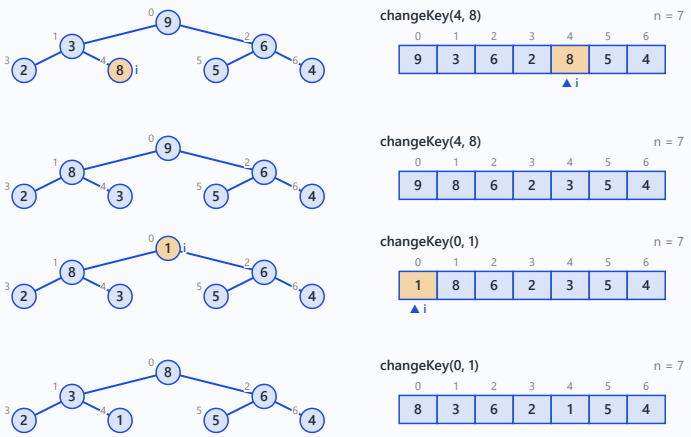


Figure 11.23: Top, increasing a key: (a) `heap[4] = 8`; (b) 8 rises a level. Bottom, decreasing a key: (c) `heap[0] = 1`; (d) 1 sinks two levels and 8 rises to the root.

1 sank two levels and 8 rose to the root. Decreasing the root's key and calling `sink(0)` is exactly the down-heap of `removeMax`, except that the value at the root is the user's new key rather than the last item.

Either way the cost is bounded by the path length, *O(log n)*; without the index, finding the item would raise it to *O(n)*.

11.5 Building a heap and heap sort

So far, items have entered the heap one at a time. When all of them are available up front (numbers read from a file, tasks waiting in an array), a faster method rearranges the array in place. It is also the first step of heap sort.

11.5.1 Two ways to build a heap from n items

Method 1: n insertions. Insert the items one by one, each with an *O(log n)* up-heap, for *O(n log n)* in total. When items arrive over time and the input can't be seen in advance (the online case), this is the only option.

Method 2: bottom-up construction. An array holding all the items already is a complete binary tree; only heap order is missing. Establish it by applying down-heap to each node from the last internal node back to the root:

```
for (int k = n / 2 - 1; k >= 0; k--)   // last internal node to root
    sink(k);                             // subtrees are heaps; sink k
```

The loop starts at $\lfloor n/2 \rfloor - 1$, the last node with a child; every later index is a leaf, since $2i + 1 \geq n$ for $i \geq \lfloor n/2 \rfloor$. Leaves are already heaps, so half the array is never touched.

Correctness rests on the loop's direction. When `sink(k)` is called, both subtrees of `k` are already heaps, because their roots have larger indices and were processed earlier or are leaves. That is exactly the situation down-heap repairs, so when the loop reaches the root, the whole tree is a heap.

> **Common mistake**
>
> Running the loop from the root, `for (int k = 0; k < n / 2; k++)`, calls `sink(k)` before the subtrees are heaps, breaking down-heap's assumption. On `[1, 2, 3, 4]` it yields `[3, 4, 1, 2]`, where 4 exceeds its parent 3. The loop must run with `k` decreasing.

11.5.2 Code trace: buildHeap()

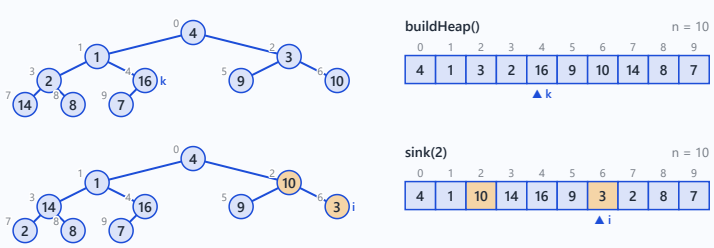
Added to the max-heap class, bottom-up construction turns the `n` items in the array into a heap:

```
public void buildHeap() {       // arbitrary array
  for (int k = n / 2 - 1; k >= 0; k--)
    sink(k);                    // subtrees are heaps
}
private void sink(int i) {        // down-heap
  while (2 * i + 1 < n) {       // left child exists?
    int j = 2 * i + 1;            // j: larger child
    if (j + 1 < n && heap[j + 1] > heap[j]) j++;
    if (heap[i] >= heap[j]) return;
    swap(i, j);  i = j;           // down one level
  }
}
```

The input is the classic example from Cormen and his coauthors: `[4, 1, 3, 2, 16, 9, 10, 14, 8, 7]`, `n = 10`. It is not a heap; the root, 4, is smaller than the 16 below it, for instance. Indices 5 through 9 are leaves, so the loop starts at $k = 10/2 - 1 = 4$.

`k`	`i`	**Children → `j`**	**Decision**	**Array**
4	4	only 9 (7) → `j = 9`	16 ≥ 7: `return`	[4, 1, 3, 2, 16, 9, 10, 14, 8, 7]
3	3	7 (14), 8 (8) → `j = 7`	2 < 14: swap	[4, 1, 3, 14, 16, 9, 10, 2, 8, 7]
3	7	`15 ≥ 10`	leaf	—
2	2	5 (9), 6 (10) → `j = 6`	3 < 10: swap	[4, 1, 10, 14, 16, 9, 3, 2, 8, 7]
2	6	`13 ≥ 10`	leaf	—
1	1	3 (14), 4 (16) → `j = 4`	1 < 16: swap	[4, 16, 10, 14, 1, 9, 3, 2, 8, 7]
1	4	only 9 (7) → `j = 9`	1 < 7: swap	[4, 16, 10, 14, 7, 9, 3, 2, 8, 1]
1	9	`19 ≥ 10`	leaf	—
0	0	1 (16), 2 (10) → `j = 1`	4 < 16: swap	[16, 4, 10, 14, 7, 9, 3, 2, 8, 1]
0	1	3 (14), 4 (7) → `j = 3`	4 < 14: swap	[16, 14, 10, 4, 7, 9, 3, 2, 8, 1]
0	3	7 (2), 8 (8) → `j = 8`	4 < 8: swap	[16, 14, 10, 8, 7, 9, 3, 2, 4, 1]
0	8	`17 ≥ 10`	leaf	—

Table 11.18: `buildHeap()` on `[4, 1, 3, 2, 16, 9, 10, 14, 8, 7]`. Each row is one pass of `sink`.



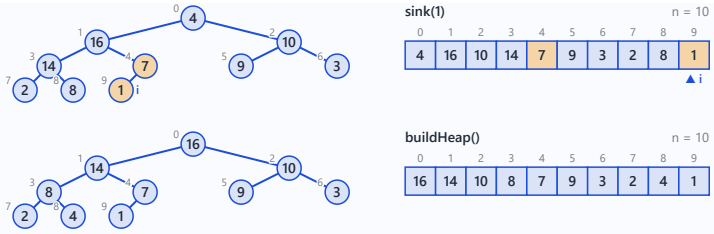


Figure 11.24: (a) The arbitrary array. (b) After `k = 4, 3, 2`. (c) After `k = 1`: 1 has sunk two levels. (d) After `k = 0`: the max-heap [16, 14, 10, 8, 7, 9, 3, 2, 4, 1].

There were seven swaps. Upper nodes traveled farther (the 4 at `k = 0` sank three levels) but are few; lower nodes were leaves or made at most one swap. That balance keeps the total linear.

11.5.3 Why is bottom-up construction O(n)?

At first sight, bottom-up construction costs *O(n log n)*: about *n/2* down-heaps of up to *O(log n)* each. That bound is true but loose. A down-heap costs at most the node's *height*, its distance to the deepest leaf below it, and most nodes are near the bottom, where heights are small.

Take a perfect binary tree of 15 nodes (all levels full, height 3).

Node height	Number of nodes	Max swaps	Total
0 (leaves)	8	0	0
1	4	1	4
2	2	2	4
3 (root)	1	3	3
n = 15			**11**

Table 11.19: Worst-case number of swaps in bottom-up construction of a 15-node tree.

Over half the nodes are leaves and do nothing, and only a few high nodes do much. The 11 swaps are fewer than the 15 nodes; a Java run on the worst input for a max-heap, 1 to 15 in increasing order, also counts exactly 11.

Building the same tree by 15 insertions costs far more, because up-heap costs depend on *depth*, the distance from the root, and most nodes are deep. With increasing input, every item climbs to the root: 2 nodes at depth 1, 4 at depth 2, and 8 at depth 3 make $0 + 2 \cdot 1 + 4 \cdot 2 + 8 \cdot 3 = 34$ swaps, the count Java gives too.

Theorem 11.3 (Cost of bottom-up construction)

Bottom-up construction of a heap from *n* items makes at most *n* swaps and runs in *O(n)* time.

Proof. A complete tree of n nodes has at most $\lceil n / 2^{h+1} \rceil$ nodes of height h: about half are leaves, a quarter are one level up, and the count keeps halving. A down-heap from height h makes at most h swaps, so the total is at most $\Sigma_{h \geq 0} (n / 2^{h+1}) \cdot h = n \cdot \Sigma_{h \geq 0} h / 2^{h+1}$. The series $1/4 + 2/8 + 3/16 + 4/32 + \ldots$ sums to exactly 1, so there are at most n swaps, each with at most two comparisons: $O(n)$ work. ■

To sum the series, let $S = 1/4 + 2/8 + 3/16 + \ldots$. Then $S/2 = 1/8 + 2/16 + \ldots$ and $S - S/2 = 1/4 + 1/8 + 1/16 + \ldots = 1/2$, so $S = 1$.

In a sentence: down-heap costs depend on height and up-heap costs on depth, and most nodes are low, with small height and large depth. Bottom-up construction works in the cheap direction.

Common mistake

Reasoning "$n/2$ nodes at $O(\log n)$ each, so $O(n \log n)$" wrongly puts bottom-up construction on a par with repeated insertion. That is only an upper bound; most nodes travel short paths, and the true total is linear.

11.5.4 Heap sort

A heap also yields a sorting algorithm. Repeatedly removing the maximum produces the items in decreasing order, and since `removeMax` parks each removed root just past the end of the heap, they collect at the end of the same array.

Heap sort has two phases:

1. **Build the heap** bottom-up: $O(n)$.
2. **Move the largest to the end, $n - 1$ times.** Swap the root with the heap's last item, shrink the heap by one, and down-heap from the root: $O(\log n)$ per round, $O(n \log n)$ in total.

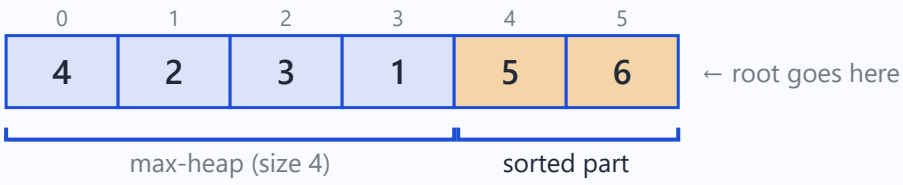


Figure 11.25: `heapSort([6, 5, 3, 4, 1, 2])` after two rounds: [4, 2, 3, 1 | 5, 6]. The first four cells are the heap, the last two the sorted part.

Mid-run (Figure 11.25), the array has two parts: a max-heap in front, shrinking by one each round, and a sorted part behind, growing by one. Every sorted item is at least as large as everything in the heap, so it is in its final place, and no extra array is needed.

Heap sort has four notable properties:

- **In place.** The heap lives in the array being sorted, so extra memory is *O(1)*.
- ***O(n log n)* in the worst case.** Quick sort is very fast on average but degrades to $O(n^2)$ with bad pivots; heap sort has no bad case.
- **Not stable.** Long-distance swaps between the root and the end can reorder equal keys.
- **Direction.** A max-heap sorts in increasing order, since the largest item goes last; a min-heap sorts in decreasing order.

11.5.5 Code: heapSort()

```
public static void heapSort(int[] a) {
  int n = a.length;
  // 1) build: from the last internal node to the root
  for (int k = n / 2 - 1; k >= 0; k--)
    sink(a, k, n);
  // 2) sort: move the largest to the end
  for (int last = n - 1; last > 0; last--) {
    swap(a, 0, last);
    sink(a, 0, last);           // heap size = last
  }
}
static void sink(int[] a, int i, int n) {
  while (2 * i + 1 < n) {
    int j = 2 * i + 1;
    if (j + 1 < n && a[j + 1] > a[j]) j++;
    if (a[i] >= a[j]) return;
    swap(a, i, j);  i = j;
  }
}
```

These are standalone `static` methods. This `sink` takes the array, an index, and the current heap size in place of the `heap` and `n` fields, but is otherwise the class's `sink`. The size must be a parameter because the heap shrinks each round while the array does not.

The first loop is bottom-up construction. In the second, `last` indexes the heap's last item. Each round swaps the root with `a[last]`, fixing the largest item in place, and calls `sink(a, 0, last)`: the new heap size is `last`, so `sink` ignores index `last` and beyond. The final round has `last = 1`; a one-item heap needs no work.

The helper `swap(a, i, j)` exchanges two cells:

```
static void swap(int[] a, int i, int j) {
  int t = a[i];  a[i] = a[j];  a[j] = t;
}
```

Heap sort turns the classic array `{4, 1, 3, 2, 16, 9, 10, 14, 8, 7}` into `{1, 2, 3, 4, 7, 8, 9, 10, 14, 16}`; Table 11.20 shows each round of the second phase.

Phase	Array
after construction	[16, 14, 10, 8, 7, 9, 3, 2, 4, 1]
`last = 9`	[14, 8, 10, 4, 7, 9, 3, 2, 1 | 16]
`last = 8`	[10, 8, 9, 4, 7, 1, 3, 2 | 14, 16]
`last = 7`	[9, 8, 3, 4, 7, 1, 2 | 10, 14, 16]
`last = 6`	[8, 7, 3, 4, 2, 1 | 9, 10, 14, 16]
`last = 5`	[7, 4, 3, 1, 2 | 8, 9, 10, 14, 16]
`last = 4`	[4, 2, 3, 1 | 7, 8, 9, 10, 14, 16]
`last = 3`	[3, 2, 1 | 4, 7, 8, 9, 10, 14, 16]
`last = 2`	[2, 1 | 3, 4, 7, 8, 9, 10, 14, 16]
`last = 1`	[1 | 2, 3, 4, 7, 8, 9, 10, 14, 16]

Table 11.20: Heap sort on `{4, 1, 3, 2, 16, 9, 10, 14, 8, 7}`. The sorted part is right of the bar.

11.5.6 Code trace: heapSort([6, 5, 3, 4, 1, 2])

A smaller input shows each `sink` call of the second phase: `a = [6, 5, 3, 4, 1, 2]`.

Phase 1. `sink` runs for `k = 2, 1, 0`, but the array is already a max-heap (6 ≥ 5, 3; 5 ≥ 4, 1; 3 ≥ 2), so no swaps are made.

`last`	Root ↔ `a[last]`	`sink(a, 0, last)`	Array after the round
5	6 ↔ 2: [2, 5, 3, 4, 1 | 6]	`i = 0`: larger child 5, swap; `i = 1`: larger child 4, swap; index 3 is a leaf	[5, 4, 3, 2, 1 | 6]
4	5 ↔ 1: [1, 4, 3, 2 | 5, 6]	`i = 0`: larger child 4, swap; `i = 1`: only child 2, swap; index 3 is a leaf	[4, 2, 3, 1 | 5, 6]
3	4 ↔ 1: [1, 2, 3 | 4, 5, 6]	`i = 0`: larger child 3, swap; index 2 is a leaf	[3, 2, 1 | 4, 5, 6]
2	3 ↔ 1: [1, 2 | 3, 4, 5, 6]	`i = 0`: only child 2, swap; index 1 is a leaf	[2, 1 | 3, 4, 5, 6]
1	2 ↔ 1: [1 | 2, 3, 4, 5, 6]	heap size 1, the root has no children	[1 | 2, 3, 4, 5, 6]
0	loop condition `last > 0` is false		[1, 2, 3, 4, 5, 6]

Table 11.21: The second phase of `heapSort([6, 5, 3, 4, 1, 2])`.

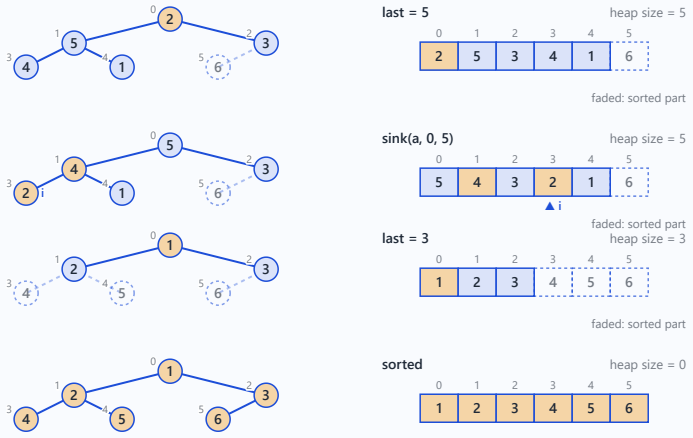


Figure 11.26: (a) `last = 5`: 6 is in its final place; heap size 5. (b) After down-heap: [5, 4, 3, 2, 1 | 6]. (c) `last = 3`: 4 is in place too. (d) Result: [1, 2, 3, 4, 5, 6].

Each round moves one cell from the heap to the sorted part. Besides the 5 root swaps, `sink` made 6 swaps, going two levels deep early on and less as the heap shrank. The total is at most $log_2\ n + log_2(n - 1) + \ldots \le n\ log_2\ n$, so $O(n\ log\ n)$.

Example 11.6

A minimal example shows that heap sort is unstable. Take `[2a, 2b, 1]`, where 2a and 2b are distinct items with equal keys, 2a first.

1. **Build.** `k = 0`: the larger child is 2b, and 2a ≥ 2b, so nothing changes.
2. `last = 2`. Swap 2a with `a[2] = 1`: `[1, 2b | 2a]`. `sink`: 1 < 2b, swap: `[2b, 1 | 2a]`.
3. `last = 1`. Swap 2b with `a[1] = 1`: `[1 | 2b, 2a]`.
4. The result, `[1, 2b, 2a]`, has its keys in order but the two equal items reversed.

11.5.7 Sorting with a priority queue

Heap sort is a special case of a general idea: any priority queue can sort, by inserting everything and then calling `removeMin` n times. The algorithm you get, and its running time, depend on the implementation.

Implementation	n × insert	n × removeMin	Resulting algorithm
Unsorted list	O(n)	$O(n^2)$	Selection sort, $O(n^2)$
Sorted list	$O(n^2)$	O(n)	Insertion sort, $O(n^2)$
Binary heap	O(n log n)	O(n log n)	Heap sort, O(n log n)

Table 11.22: Sorting with a priority queue: which algorithm does each implementation give?

With an unsorted list, each `removeMin` selects the smallest remaining item: selection sort. With a sorted list, each `insert` puts the item in place within the sorted part: insertion sort. A heap balances the two phases.

This is the distinction between ADT and data structure in miniature: the algorithm ("insert everything, then remove everything") never changes, yet changing the implementation cuts the time from $O(n^2)$ to $O(n \log n)$.

11.5.8 Other kinds of heaps

Other heaps suit other mixes of operations.

In a **d-ary heap**, a complete tree whose nodes have up to d children, the 0-based array formulas give node `i` children $di + 1$ through $di + d$ and parent $\lfloor (i - 1)/d \rfloor$. In a 3-ary heap, index 7 has parent $\lfloor 6/3 \rfloor = 2$, whose children are 7, 8, and 9.

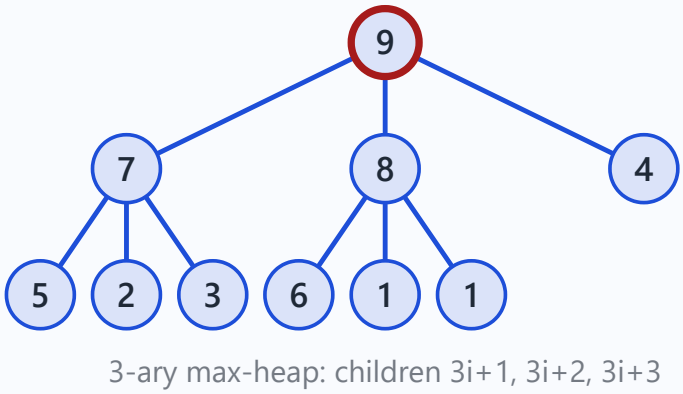


Figure 11.27: A 3-ary max-heap: root 9; children 7, 8, 4; 7 has children 5, 2, 3, and 8 has 6, 1, 1.

Its height is $\log_d n$, shrinking as d grows. Up-heap makes one comparison per level, so insertion speeds up to $O(\log_d n)$, but down-heap must find the largest of d children per level, so removal costs $O(d \log_d n)$. With many insertions and key decreases but few removals, as in Dijkstra's algorithm on dense graphs, a d-ary heap can beat a binary one.

Binomial and **Fibonacci heaps** are forests rather than single trees, designed chiefly to speed up *merging* two queues and decreasing keys. Fibonacci heaps appear in the theoretically fastest versions of Dijkstra's and Prim's algorithms.

Min-heap type	insert	removeMin	decrease key	merge
Binary heap	O(log n)	O(log n)	O(log n)	O(n)
d-ary heap	$O(\log_d n)$	$O(d \log_d n)$	$O(\log_d n)$	O(n)
Binomial heap	O(log n)	O(log n)	O(log n)	O(log n)

Min-heap type	insert	removeMin	decrease key	merge
Fibonacci heap	O(1)	O(log n)*	O(1)*	O(1)

Table 11.23: Operation costs of min-heap variants. * amortized.

Merging binary heaps costs *O(n)*: concatenate the arrays and rebuild bottom-up. Despite the Fibonacci heap's theoretical edge, the binary heap usually wins in practice, thanks to simple code, small constants, and a cache-friendly array. Java's `PriorityQueue` and Python's `heapq` both use one.

11.6 Java and applications

Writing a heap yourself is the best way to understand it, but real programs usually use the built-in class, `java.util.PriorityQueue` in Java. This section covers its use, its internals, and typical applications of heaps.

11.6.1 Java's built-in PriorityQueue

```
PriorityQueue<Integer> pq = new PriorityQueue<>();
pq.offer(5); pq.offer(1); pq.offer(8);
pq.peek();   // 1  (min-heap, O(1))
pq.poll();   // 1  (O(log n))

// reversed comparator for a max-heap
PriorityQueue<Integer> maxPq =
    new PriorityQueue<>(Comparator.reverseOrder());

// patients: smaller priority first
PriorityQueue<Patient> er = new PriorityQueue<>(
    Comparator.comparingInt(p -> p.priority));
```

`PriorityQueue<Integer>` is a **min-heap** by default: it orders items naturally and keeps the smallest in front. `offer` inserts, `peek` looks at the front item, and `poll` removes it. After three `offer` calls, `peek()` returns 1 without changing the queue, and `poll()` returns 1 and removes it. On an empty queue these two return `null`, while their counterparts `add`, `element`, and `remove` throw an exception.

A `Comparator` passed to the constructor changes the ordering. `Comparator.reverseOrder()` makes the queue act as a max-heap: with the same 5, 1, 8, `peek()` returns 8. For our own classes, an expression like `Comparator.comparingInt(p -> p.priority)` names the field to compare. If Alan (3), Eve (1), Carl (2), and Nina (1) enter the `er` queue and are removed in turn, the two priority-1 patients leave first, then Carl, then Alan.

Four facts about the class are worth knowing:

- **Structure.** An array-based binary min-heap whose array grows automatically.
- **Costs.** `offer` and `poll` are *O(log n)*; `peek` and `size` are *O(1)*; `remove(object)` and `contains` are *O(n)*, since they search the array linearly.
- **Concurrency.** The class is not thread-safe; when threads share a queue, use `PriorityBlockingQueue`.
- **Ties.** Equal priorities have no FIFO guarantee. Eve leaving before Nina is an accident of this input; another insertion order could reverse them.

> **Common mistake**
>
> `PriorityQueue` is not a max-heap by default. Its default is natural order, smallest first; for largest first, pass `Comparator.reverseOrder()` to the constructor.

11.6.2 Code trace: what PriorityQueue does inside

The program below shows the class from outside; the trace shows its internal array (named `queue` in the Java source) at each step.

```
PriorityQueue<Integer> pq =
    new PriorityQueue<>();        // min-heap
pq.offer(5);
pq.offer(1);
pq.offer(8);
System.out.println(pq.peek());    // looks
System.out.println(pq.poll());    // removes
System.out.println(pq.poll());
System.out.println(pq.size());
```

Call	What happens inside?	`queue`	Console
`new PriorityQueue<>()`	empty array	[ ]	—
`offer(5)`	5 becomes the root	[5]	—
`offer(1)`	added at index 1; parent 5 > 1, so `siftUp` moves it to the root	[1, 5]	—
`offer(8)`	added at index 2; parent 1 ≤ 8, no up-heap	[1, 5, 8]	—
`peek()`	root returned, not removed	[1, 5, 8]	1
`poll()`	root 1 taken; last item 8 to the root; `siftDown`: 8 swaps with the smaller child, 5	[5, 8]	1
`poll()`	root 5 taken; last item 8 to the root; one item, no `siftDown`	[8]	5

Call	What happens inside?	`queue`	Console
`size()`	counter read	[8]	1

Table 11.24: The internal array of `PriorityQueue`. The Console column shows what the program prints.

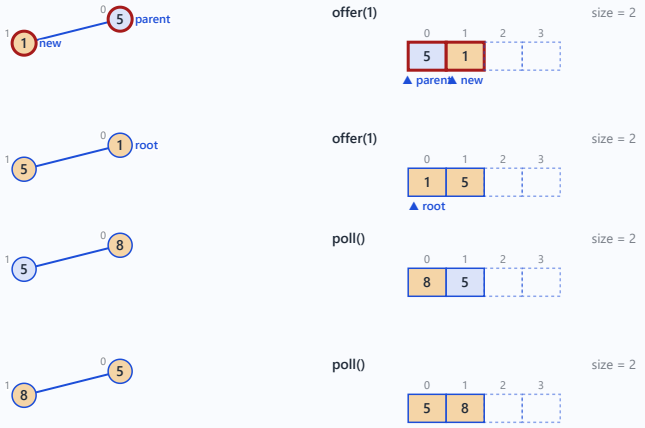


Figure 11.28: (a) `offer(1)`: 1 at index 1 violates min-heap order. (b) After `siftUp`: [1, 5]. (c) `poll()`: the last item, 8, moves to the root. (d) After `siftDown`: [5, 8].

The program prints 1 (`peek`), 1 and 5 (`poll` twice), and 1 (`size`). Java's `siftUp` and `siftDown` are min-heap versions of our `swim` and `sink`, generalized to work with any `Comparator`.

Printing a `PriorityQueue` shows its internal array, not a sorted list: inserting 3, 1, 2 prints `[1, 3, 2]`. A `for-each` loop visits the same internal order. Only repeated `poll` calls yield the items in sorted order.

Common mistake

Believing that "a heap is sorted" leads people to expect sorted output from printing or iterating over a `PriorityQueue`. Only the root's position is fixed; the rest merely satisfy heap order, and `System.out.println(pq)` prints the array in heap order.

11.6.3 The kth largest element

Finding the kth largest element is a common task: the third-highest exam grade, the tenth-busiest hour on a server. Sorting works but takes *O(n log n)* and keeps all *n* items in memory. A min-heap of at most *k* items does better:

```
static int kthLargest(int[] a, int k) {
  PriorityQueue<Integer> pq = new PriorityQueue<>();
  for (int x : a) {
    pq.offer(x);                    // O(log k)
    if (pq.size() > k) pq.poll();  // drop the smallest
  }
  return pq.peek();       // smallest of the k largest
}
```

The invariant: after each step, the heap holds the *k* largest items seen so far, with the smallest of them at the root. Each new item is added, and if the heap reaches *k + 1* items its minimum is dropped, since an item with *k* larger ones can't be among the *k* largest. At the end, the root is the smallest of the *k* largest: the kth largest.

Here is `kthLargest([5, 1, 9, 3, 7, 6], 3)`.

`x`	**After** `offer(x)`	`size > k` **?**	**Dropped**	**Heap after the round**
5	[5]	1 > 3? no	—	[5]
1	[1, 5] (1 moved to the root)	2 > 3? no	—	[1, 5]
9	[1, 5, 9]	3 > 3? no	—	[1, 5, 9]
3	[1, 3, 9, 5] (3 swapped with 5)	4 > 3? yes	1	[3, 5, 9]
7	[3, 5, 9, 7]	yes	3	[5, 7, 9]
6	[5, 6, 9, 7] (6 swapped with 7)	yes	5	[6, 7, 9]
`peek()`				returns 6

Table 11.25: `kthLargest([5, 1, 9, 3, 7, 6], 3)`. The heap columns show the internal array.

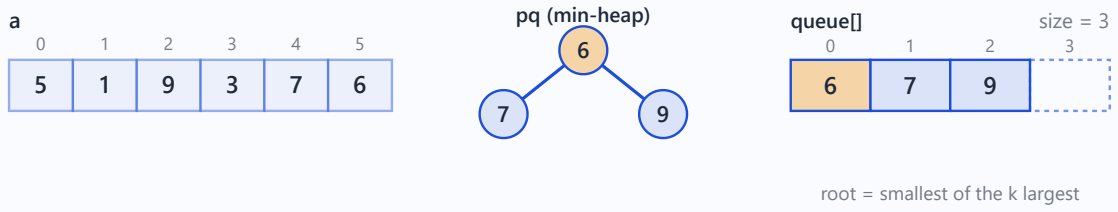


Figure 11.29: At the end, the heap holds the three largest items (6, 7, 9); the 6 at the root is the third largest.

The dropped items, 1, 3, and 5, are all outside the top three; sorted, the array is [1, 3, 5, 6, 7, 9], and the third largest is indeed 6.

Cost: with at most *k + 1* items in the heap, each `offer` and `poll` is *O(log k)*, for *O(n log k)* time and *O(k)* extra memory, much better than sorting when *k* is small. The memory saving matters too: items streaming from a file or network need not all be stored.

Common mistake

A max-heap is the wrong tool for the kth largest element: it must hold all the items and be polled k times, using $O(n)$ memory. A min-heap capped at k keeps the answer at its root and discards small items immediately.

11.6.4 Merging k sorted lists

Heaps also merge k sorted lists into one, as in external sorting, which sorts a large file in pieces and then merges them. Let N be the total number of items.

Put the first item of each list into a min-heap, each tagged with its list. Then, until the heap is empty, remove the minimum, output it, and insert the next item of its list, if any. The heap never holds more than one item per list, so at most k.

Example 11.7

Three sorted lists: A = [1, 7], B = [2, 3, 9], C = [4]. Heap items are written "value(list)."

1. Start: heap {1(A), 2(B), 4(C)}.
2. 1(A) out, output [1]; 7 from A in: {2(B), 4(C), 7(A)}.
3. 2(B) out, output [1, 2]; 3 from B in: {3(B), 4(C), 7(A)}.
4. 3(B) out, output [1, 2, 3]; 9 from B in: {4(C), 7(A), 9(B)}.
5. 4(C) out; C is exhausted: {7(A), 9(B)}. Then 7 and finally 9.
6. Result: [1, 2, 3, 4, 7, 9].

Each item enters and leaves the heap once, at $O(\log k)$ per operation, so the total is $O(N \log k)$, against $O(N \log N)$ for gathering everything into one array and sorting.

11.6.5 A priority scheduler

An operating system's scheduler decides which waiting process gets the CPU next. In priority scheduling each process has a priority level, and a common design keeps one FIFO queue per level.

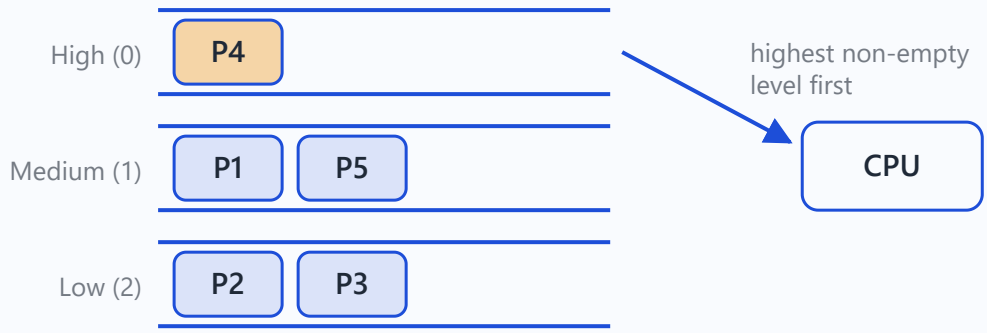


Figure 11.30: Three FIFO queues at three priority levels: P4 at high, P1 and P5 at medium, P2 and P3 at low. The CPU picks P4 first.

- **Strict priority.** The CPU always takes the front process of the highest non-empty queue. In Figure 11.30, P4 runs first; once the high level is empty, P1 and P5 get their turn.
- **FIFO within a level.** Equal-priority processes run in arrival order. If each runs for a time slice (quantum) and then rejoins the back of its queue, the level is scheduled round robin.
- **Starvation and aging.** If high-priority processes keep arriving, low-priority ones may never run: **starvation**. The usual cure is **aging**, raising a waiting process's priority step by step until it reaches the upper levels and runs.

With few levels, one queue per level is simplest and fastest, since finding the highest non-empty level takes constant time. With a wide range of priorities (say, a deadline per process), a single heap is used instead, with (priority, arrival number) keys to keep ties FIFO. Aging then becomes a key change, *O(log n)* in an adaptable priority queue.

11.7 Common mistakes

The table collects the most frequent mistakes from this chapter; use it as a checklist when reviewing heap code.

Mistake	What happens?	The fix
"A heap is sorted"	After inserting 3, 1, 2, `println(pq)` prints `[1, 3, 2]`; code expecting sorted output fails.	Only the root's position is fixed. For sorted output, `poll` the items one by one.
Index formulas	Mixing 0- and 1-based formulas links nodes to the wrong parents and children.	0-based: *2i + 1*, *2i + 2*, *(i − 1)/2*. 1-based: *2i*, *2i + 1*, *i/2*.
Right-child bound	Without `j + 1 < n`, a deleted item outside the heap is read, or the index runs off the array.	`if (j + 1 < n && heap[j + 1] > heap[j]) j++;`
Wrong child	Down-heap in a max-heap swaps with the smaller child and breaks the order.	Larger child in a max-heap, smaller in a min-heap.
Searching a heap	`contains` and `remove(object)` are *O(n)* and slow down code that uses them often.	Keep location-aware entries or a separate map.

Mistake	What happens?	The fix
Direction mix-up	`PriorityQueue` is expected to return the largest item.	The default is a min-heap; for max, use `Comparator.reverseOrder()`.
Build loop direction	Calling `sink` from the root toward the leaves may not produce a heap.	Run `k` down from $n/2 - 1$ to 0.
Expecting FIFO on ties	Equal-priority items may leave out of insertion order.	Use (priority, arrival number) keys.

Table 11.26: Common mistakes in heap and priority queue code.

Most of these bugs hide in small tests: the right-child bug appears only at certain sizes, the tie bug only with equal keys. To test a heap reliably, check the array with `isHeap` after every operation and compare results against a library class such as `PriorityQueue` on random inputs.

11.8 Summary

- A **priority queue** is an ADT whose items leave by priority, not arrival. Items are (key, value) pairs; the operations are `insert`, `min`, `removeMin`, `size`, and `isEmpty` (`max` and `removeMax` in a max-queue). Equal keys have no FIFO guarantee; add an arrival number if needed.
- **List-based implementations**: an unsorted list inserts in $O(1)$ and removes in $O(n)$, a sorted list the reverse; either way n insertions and removals take $O(n^2)$. Keep the top item at the end of a sorted array, at the head of a linked list.
- A **binary heap** satisfies heap order (parent ≥ child in a max-heap) and completeness. The top item is at the root, and the height is $\lfloor \log_2 n \rfloor$. A heap is neither sorted nor a binary search tree.
- The **array representation** stores nodes in level order, without gaps or references. With 0-based indices, the children are $2i + 1$ and $2i + 2$, the parent $\lfloor (i - 1)/2 \rfloor$.
- **Insertion** writes to `heap[n]` and applies up-heap; **removeMax** moves the last item to the root and applies down-heap toward the larger child. Both are $O(\log n)$. Down-heap must test `j + 1 < n` for the right child. Doubling the capacity makes resizing amortized $O(1)$.
- An **adaptable priority queue** adds `replaceKey` and `remove`; if items know their array positions, each takes one up-heap or down-heap, $O(\log n)$.
- **Bottom-up construction** applies down-heap from the last internal node ($\lfloor n/2 \rfloor - 1$) to the root in $O(n)$, because down-heap costs depend on height and most nodes are low; n insertions take $O(n \log n)$.
- **Heap sort** builds a heap, then $n - 1$ times moves the root to the end and shrinks the heap: in place, $O(n \log n)$ in the worst case, not stable. Sorting with a priority queue gives selection sort on an unsorted list and insertion sort on a sorted one.
- **Java's** `PriorityQueue` is an array-based binary min-heap; pass `Comparator.reverseOrder()` for a max-heap. A size-k min-heap finds the kth largest element in $O(n \log k)$ and merges k sorted lists in $O(N \log k)$.

Operation	Heap	How?
peek (max/min)	O(1)	`heap[0]`
insert	O(log n)	put at `heap[n]`, up-heap
removeMax / removeMin	O(log n)	move the last item to the root, down-heap
change key	O(log n)	up-heap if increased, down-heap if decreased (max-heap)
build heap	O(n)	down-heap from $\lfloor n/2 \rfloor - 1$ to 0
heap sort	O(n log n)	build + move the root to the end $n - 1$ times

Table 11.27: Summary of heap operations.

11.9 Key terms

Each term links to the course glossary, which gives its Turkish equivalent and its use in other chapters.

priority queue · abstract data type (ADT) · key · value · queue · FIFO · interface · generic · binary heap · heap · max-heap · min-heap · complete binary tree · perfect binary tree · root · leaf · internal node · parent · child · sibling · level · level order · height · depth · up-heap · down-heap · swap · capacity · dynamic array · amortized analysis · adaptable priority queue · bottom-up heap construction · heap sort · in-place · stable sort · selection sort · insertion sort · quick sort · round robin · aging

11.10 Test Yourself

1. An empty min-priority queue receives `insert(7, a)`, `insert(2, b)`, `insert(9, c)`, `removeMin()`, `insert(1, d)`, `min()`, `removeMin()`, `removeMin()`. What do the value-returning calls return, and what remains?

   Answer

   The first `removeMin()` returns (2, b), leaving {(7,a), (9,c)}. Then (1, d) is the smallest: `min()` returns it without removing it, and the next two `removeMin()` calls return (1, d) and (7, a). Only (9, c) remains; `size()` is 1.

2. Insert 3, 8, 6, 10, 1, 9 into an empty max-heap of capacity 4. What is the final array, how many

swaps occur, and when does the capacity grow?

Answer

[3] → 8 swaps with 3: [8, 3] → 6 stays: [8, 3, 6] → 10 swaps with 3, then 8: [10, 8, 6, 3] → the array is full, so the capacity grows to 8 before 1 goes in; 1 stays: [10, 8, 6, 3, 1] → 9 swaps with its parent 6 and stops under 10. Final array `[10, 8, 9, 3, 1, 6]`, 4 swaps.

3. Call `removeMax()` twice on `[20, 15, 18, 7, 12, 16, 4]`. What is returned, and what is the final heap?

Answer

First, 20; 4 goes to the root: [4, 15, 18, 7, 12, 16]; larger child 18: [18, 15, 4, 7, 12, 16]; at index 2 the only child is 16: [18, 15, 16, 7, 12, 4]. Second, 18; 4 to the root: [4, 15, 16, 7, 12]; larger child 16: [16, 15, 4, 7, 12], and index 2 is a leaf. Final heap `[16, 15, 4, 7, 12]`.

4. What is the height of a 25-item binary heap? Where does the bottom-up loop start, and how many nodes are leaves? With 0-based indices, where is the parent of index 20?

Answer

Height $\lfloor \log_2 25 \rfloor = 4$ ($16 \le 25 < 32$). The loop starts at $\lfloor 25/2 \rfloor - 1 = 11$; the 13 nodes at indices 12 to 24 are leaves. Index 20's parent is $\lfloor 19/2 \rfloor = 9$; its first child would be $41 \ge 25$, so 20 is a leaf.

5. Turn `[5, 13, 2, 25, 7, 17, 20, 8, 4]` into a max-heap bottom-up. What is the array after each `sink(k)`, and how many swaps occur?

Answer

$n = 9$, so `k` starts at 3. `k = 3`: 25 ≥ 8, 4, no change. `k = 2`: 2 swaps with its larger child 20: [5, 13, 20, 25, 7, 17, 2, 8, 4]. `k = 1`: 13 swaps with 25 and then stays, since 13 ≥ 8, 4: [5, 25, 20, 13, 7, 17, 2, 8, 4]. `k = 0`: 5 swaps with 25, 13, and 8 in turn: `[25, 13, 20, 8, 7, 17, 2, 5, 4]`. 5 swaps.

6. Heap-sort `[5, 2, 8, 1, 9]`. What is the array after construction and after each "root to the end + sink" round?

Answer

Construction: at `k = 1`, 2 ↔ 9; at `k = 0`, 5 ↔ 9, and 5 stops above its children 1 and 2: [9, 5, 8, 1, 2]. Rounds: [8, 5, 2, 1 | 9] → [5, 1, 2 | 8, 9] → [2, 1 | 5, 8, 9] → [1 | 2, 5, 8, 9]. Result [1, 2, 5, 8, 9].

7. Call `changeKey(1, 2)` on `[15, 9, 12, 4, 8, 10, 3]`. Which operation runs, and what is the final array?

   Answer

   The key drops from 9 to 2, so `sink(1)` runs. Index 1's children are 4 and 8; 2 < 8 (index 4), so they swap, and index 4 is a leaf. Final array `[15, 8, 12, 4, 2, 10, 3]`; the root and the right subtree are never examined.

8. What does `kthLargest([3, 11, 6, 2, 9, 14, 5], 2)` return, and which items are dropped?

   Answer

   The heap holds at most two items. After 3 and 11, the arrival of 6 drops 3: {6, 11}. 2 is dropped as soon as it enters. 9 drops 6, 14 drops 9, and 5 drops itself. The heap ends as {11, 14}, and `peek()` returns 11, the second largest. Dropped, in order: 3, 2, 6, 9, 5.

9. For a perfect binary tree of 1023 nodes (height 9), how many swaps does bottom-up construction make in the worst case? How many would inserting the same items one by one in increasing order make?

   Answer

   There are 2^{9-h} nodes of height h, each making at most h swaps: $\Sigma\, h \cdot 2^{9-h} = 1013$, which equals $n - \log_2(n + 1) = 1023 - 10$, less than n. Increasing insertion sends every item to the root, the 2^d nodes at depth d making d swaps each: $\Sigma\, d \cdot 2^d = 8 \cdot 2^{10} + 2 = 8194$, over eight times as many.

10. The adaptable queue's `remove` deletes the item at index 4 (3) from `[10, 4, 9, 1, 3, 8, 7]`. What replaces it, which repair is needed, and what is the final array?

    Answer

    It is swapped with the last item, 7, and `n` becomes 6: [10, 4, 9, 1, 7, 8]. 7 exceeds its parent 4, so up-heap is needed, not down-heap: 7 and 4 swap, and since 10 ≥ 7 it stops. Final array `[10, 7, 9, 1, 4, 8]`. The replacement came from the other branch of the tree, not the deleted node's subtree.

11.11 Further reading

- M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, *Data Structures and Algorithms in Java* (6th ed., 2014), Chapter 9: the priority queue ADT, list and heap implementations, and adaptable priority queues with location-aware entries, in Java close to these notes.

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms* (4th ed., 2022), Chapter 6: heaps, the proof that bottom-up construction is linear, heap sort, and heap-based priority queues, formally; the ten-item example here comes from this book.
- R. Sedgewick and K. Wayne, *Algorithms* (4th ed., 2011), Section 2.4: the binary heap in a 1-based array with operations named swim and sink, plus indexed priority queues and multiway merging.
- R. W. Floyd, "Algorithm 245: Treesort 3" (1964): the one-page paper that introduced bottom-up heap construction, improving Williams's heapsort.

11.12 References

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022.
2. R. W. Floyd, "Algorithm 245: Treesort 3," *Communications of the ACM*, 7(12), p. 701, 1964.
3. M. T. Goodrich, R. Tamassia, M. H. Goldwasser, *Data Structures and Algorithms in Java*, 6th ed., Wiley, 2014.
4. Oracle, *Java SE 17 API Specification*: documentation for `java.util.PriorityQueue`.
5. R. Sedgewick, K. Wayne, *Algorithms*, 4th ed., Addison-Wesley, 2011.
6. J. W. J. Williams, "Algorithm 232: Heapsort," *Communications of the ACM*, 7(6), pp. 347–348, 1964.

PART IV

Maps, Sets and Graphs

Chapters 12–14

PART IV · CHAPTER 12

Maps and Hash Tables

Nobody looks up Ayşe Demir's number by reading the phone book from page one; the name leads straight to the right place. A map makes this key-to-value lookup an abstract data type, and a hash table is its most common implementation: a hash function turns the key into an array index, so the item is found without comparisons, in constant time on average. We start with the map ADT and the limits of direct addressing, then study hash functions and Java's `equals` / `hashCode` contract. The two main collision strategies, separate chaining and open addressing, are coded and traced step by step. The last sections cover the load factor, resizing, the internals of Java's `HashMap`, and typical uses of hash tables.

Code for this chapter: `github.com/sercankulcu/data-structures-java/Bolum12/src`

LEARNING OUTCOMES

The map model
Explain key–value pairs and how a hash table works; give a map's contents and return values after a sequence of operations.

Collisions
Apply separate chaining and open addressing (linear and quadratic probing, double hashing) by hand and in code; show why deletion under open addressing needs a special marker.

Load factor
Relate the load factor, resizing, and the spread of keys over buckets; compute the expected number of probes.

Correctness and cost
Apply the `equals` / `hashCode` contract; compare and justify the average and worst-case costs of hash table operations.

12.1 Maps and the idea of hashing

So far we have reached items by position (cell 5 of an array) or in order (from the head of a list). In everyday life we look things up by *name*: the registrar finds a student by ID, a dictionary lists a meaning under the word, a compiler finds a variable's type from its name. The map ADT describes such lookups, and hashing makes them fast.

12.1.1 Maps

Each line of a phone book holds a name, which identifies the line, and a number attached to it. Asking for Ayşe Demir's number, we have the name and want the number. The name is the **key**, the number the **value**.

> **Definition 12.1**
>
> A **map** is an abstract data type that stores **key-value** pairs. Each key occurs at most once — keys are **unique** — and a value is always reached through its key.

Uniqueness is the heart of the definition. If a name appeared twice, "this person's number" would have no single answer. So inserting with an existing key creates no new pair but changes the existing pair's value. Values carry no such restriction: two people may share a number, two students a grade.

Figure 12.1 shows a phone book as a map. Three names pass through a function that sends them to slots 02, 01, and 14 of a table, where their numbers are stored. That function, which turns a name into a slot number, is the *hash function* — the real subject of this chapter.

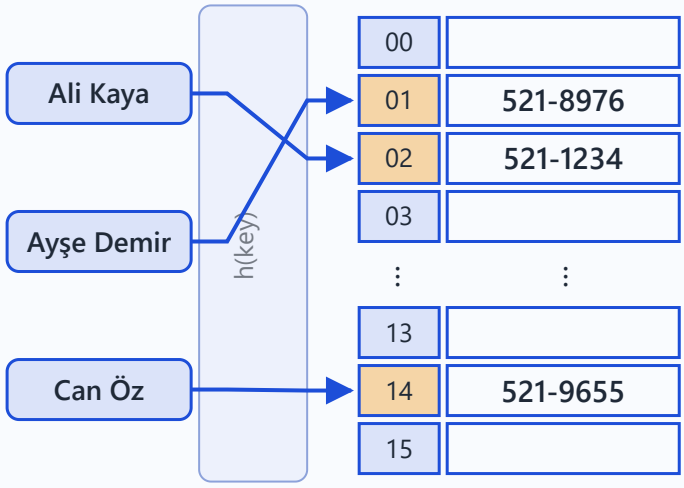


Figure 12.1: A phone book is a map. Ali Kaya, Ayşe Demir, and Can Öz pass through `h(key)` to slots 02, 01, and 14, which store their numbers; the other slots are empty.

As an *abstract data type*, a map says *what* a structure does — which operations exist and what each changes — not *how*. It can be implemented with an unsorted array, a sorted array, a search tree, or a hash table; all answer the same questions, at different speeds.

The hash table is the most common choice because insertion, search, and deletion take ***O(1)* time on average**: a key's value is reached as fast among a million items as among a thousand, while even binary search on a sorted array needs *O(log n)* comparisons. The rest of the chapter explains where this average constant time comes from and when it breaks down.

> **Common mistake**
>
> Treating "map" and "hash table" as synonyms. A map is an interface; a hash table is one structure that implements it. In Java, `Map` is an interface, implemented by `HashMap` with a hash table and by `TreeMap` with a balanced search tree. Both offer `put` / `get`, but their costs and extras (such as sorted iteration) differ.

12.1.2 The map ADT's operations

The map ADT has four operations: insertion, search, deletion, and update. Java's `java.util.Map` calls them `put`, `get`, and `remove`; a `put` with an existing key is the update.

Operation	`java.util.Map`	Description
Insertion	`put(k, v)`	Adds the pair.
Search	`get(k)`	Returns the value of `k`, or `null` if absent.
Deletion	`remove(k)`	Removes the pair with key `k` and returns its value.
Update	`put(k, v')`	If `k` is present, sets its value to `v'` and returns the old value.
Other	`size()`, `isEmpty()`, `containsKey(k)`, `keySet()`, `values()`	

Table 12.1: The map ADT's operations and their `java.util.Map` counterparts.

The snippet below uses them in turn; the comments give each result.

```
Map<Integer, String> m = new HashMap<>();
m.put(105, "Murat");   m.put(21, "Leyla");
m.put(21, "Ali");      // update: "Leyla" → "Ali"
m.get(21);             // "Ali"
m.remove(105);         // "Murat"
m.get(88);             // null: no such key
```

Line 1 creates an empty map from `Integer` keys to `String` values. The declared type is the `Map` interface and `new HashMap<>()` its hash table implementation — a good habit, since switching to `TreeMap` later changes only this line.

Line 2 adds (105, "Murat") and (21, "Leyla"). Line 3 adds nothing: key 21 exists, so its value changes from "Leyla" to "Ali" and the map still holds two pairs. `m.get(21)` now returns "Ali", `m.remove(105)` removes that pair and returns "Murat", and `m.get(88)` returns `null` because 88 was never a key.

`put` also returns a value, ignored above: `null` for a new key, the *old* value after an update. This is the easiest way for code to tell the two apart.

Call	Map (after)	`size()`	Returns
(start)	{ }	0	—
`put(105, "Murat")`	{105=Murat}	1	`null` (new key)
`put(21, "Leyla")`	{105=Murat, 21=Leyla}	2	`null` (new key)
`put(21, "Ali")`	{105=Murat, 21=Ali}	2	"Leyla" (old value)
`get(21)`	unchanged	2	"Ali"
`remove(105)`	{21=Ali}	1	"Murat"
`get(88)`	unchanged	1	`null` (no key)

Table 12.2: Trace of the snippet, including what `put` returns.

Common mistake

Reading a `null` from `get` as "the key is absent." `HashMap` allows `null` values: after `m.put(7, null)`, `m.get(7)` returns `null` although 7 is in the map. To test for a key, use `containsKey(k)`.

12.1.3 Constant-time access in an array

Can an array be searched in *O(1)* time? Not by comparing items. In an unsorted array the target could be anywhere, and **linear search** may check every cell, in *O(n)*. In a sorted array **binary search** halves the range each step, in *O(log n)*, and no comparison-based method does better in general.

But if we know an item's *position* — its index — access takes one step:

```
int[] sortedArr = {1, 3, 5, 7, 9};
int idx = 2;                    // index of the item we want
int item = sortedArr[idx];      // O(1): item = 5
```

`sortedArr[idx]` scans nothing: it adds *index × cell size* to the array's start address and goes straight there, in the same time for five items or five million. Indexed access is *O(1)*.

Hence the idea behind the hash table: **if a key can be turned directly into an array index, we reach the right cell in one step, with no comparisons.** The rest of the chapter shows how, and what gets in the way.

12.1.4 Indexing: the key is the index

Take the simplest case, integer keys from 0 to 9. Key *k* goes in cell *k* — 3 in `A[3]`, 7 in `A[7]` — and to find a key we look only at its cell. This is **indexing**.

Figure 12.2 shows the key space {1, 3, 5, 6, 7, 9}: each key sits in the cell with its own number, and cells 0, 2, 4, and 8 are empty.

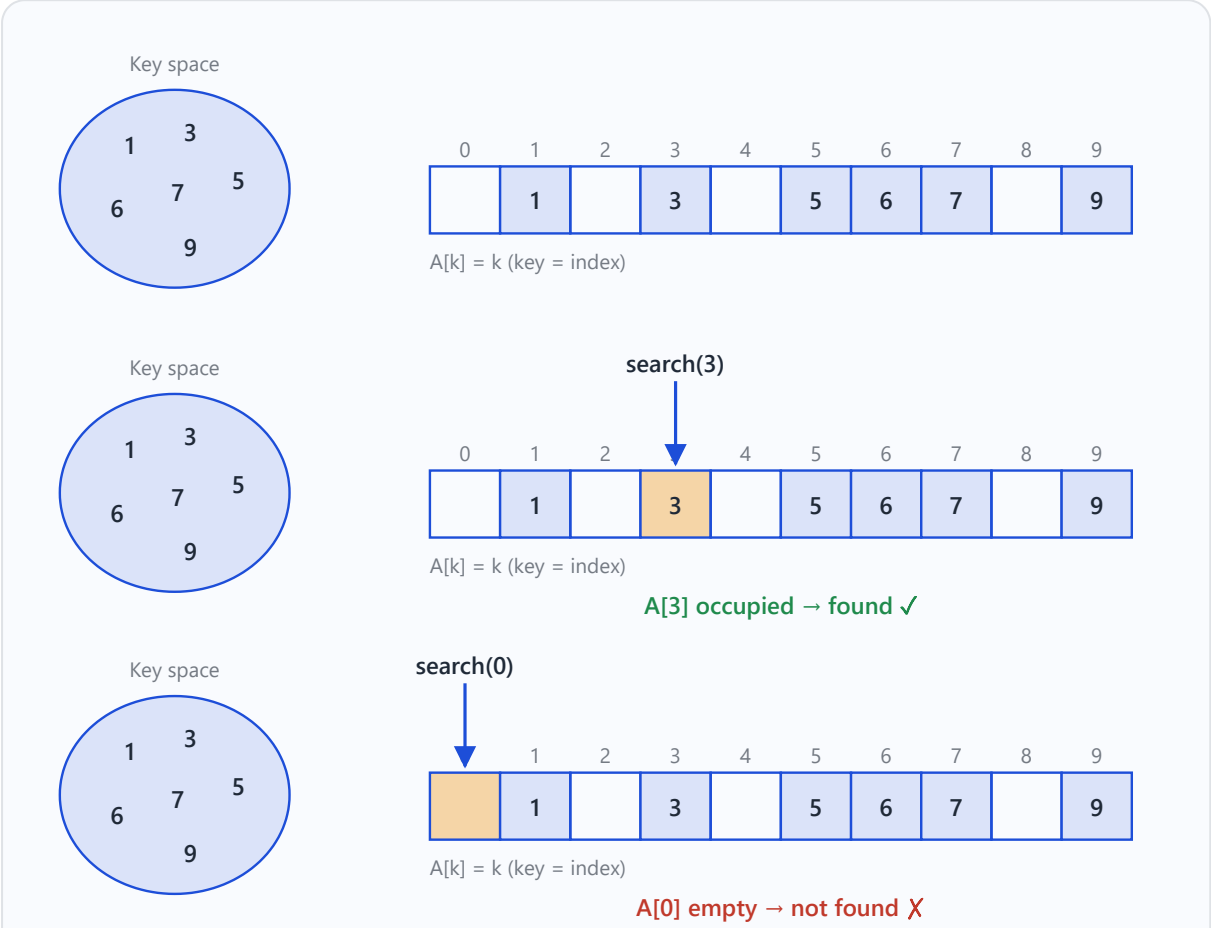


Figure 12.2: Indexing. Top: each *k* in the key space goes in `A[k]`. Middle: `search(3)` looks at `A[3]`, finds it occupied, and finds 3. Bottom: `search(0)` finds `A[0]` empty, so 0 is absent. Both take one step.

A successful and an unsuccessful search cost the same: each looks at a single cell. Insertion and deletion likewise change one cell.

12.1.5 Direct addressing and its limits

Generalizing indexing gives **direct addressing**: one cell for every possible key, with key *k* in `A[k]`. Search, insertion, and deletion are *O(1)*, with no comparisons at all.

Figure 12.3 shows the catch. Ali, Beyza, Deniz, and Onur occupy cells 0, 2, 5, and 29 of a 30-cell array: four cells used, thirty allocated, 26 empty but still taking memory.

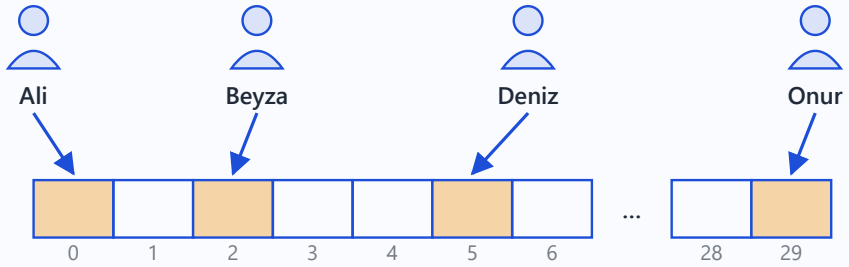


Figure 12.3: Direct addressing. Four people occupy cells 0, 2, 5, and 29 of a 30-cell array; since every possible key has a cell, the other 26 stay empty.

Direct addressing has two serious limits:

- **Wasted memory.** A large key space needs a large array. Nine-digit student IDs need 10^9 cells — about 4 GB even at 4 bytes per `int` — for a department of a few hundred students, using less than one cell in a million.
- **Keys need not be integers.** A string such as "Ali" or "Beyza" cannot index an array; there is no `A["Ali"]`.

The way out is a function that *compresses* the large, possibly non-integer key space into a small array, mapping every key to an index from 0 to $N - 1$, where the array size N is proportional to the number of items stored. Such a function is a **hash function**.

Example 12.1

A faculty stores the GPAs of 800 students with nine-digit IDs, each GPA an 8-byte `double`. Compare the memory needed by direct addressing and by a hash table.

1. **Direct addressing:** 10^9 possible IDs, so $10^9 \times 8$ bytes, about 8 GB.
2. **Hash table:** twice as many buckets as students is plenty, $N \approx 1600$. Even storing the ID (4 bytes) with the GPA (8 bytes) in each bucket takes $1600 \times 12 = 19{,}200$ bytes, under 20 KB.
3. **Result:** both run in constant time on average, but the hash table uses about 400,000 times less memory. The price is the chance of two IDs sharing a bucket, and managing that chance is what this chapter is about.

12.1.6 The hashing technique

Hashing stores items so they can be reached quickly. The key step is to find an item's position in the array fast; that position is a **bucket**, or under open addressing a **slot**. Once it is known, inserting, removing, or updating the item touches only that cell.

Definition 12.2

A **hash function** h maps each key to an integer from 0 to $N - 1$. An array of N buckets with a hash function is a **hash table**; the item with key k is stored in bucket $h(k)$.

In Figure 12.4, the key "Ali" goes into the hash function, which produces 8, and the item is placed in bucket 8. Looking "Ali" up later repeats the calculation, gets 8 again, and goes straight to bucket 8.

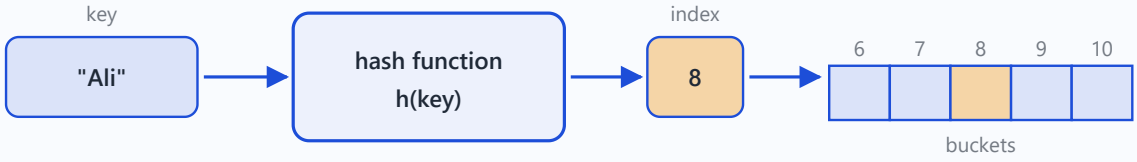


Figure 12.4: Hashing. The key "Ali" passes through h and yields index 8; the item goes into bucket 8.

Where does the 8 come from? Every Java object has a `hashCode()` method; for "Ali" it returns 65918. With 13 buckets we take the remainder: $65918 \bmod 13 = 8$. A "hash function" is thus two steps — key to integer, integer to the table's range — both examined in the next section, along with how 65918 is computed.

Common mistake

Assuming a hash table keeps its items in order. Items are scattered over the buckets the hash function picks, regardless of key size or insertion order: "Ali" in bucket 8 may precede a key in bucket 2 alphabetically. So a hash table cannot quickly answer "smallest key" or "list in sorted order."

12.2 Hash functions

The hash function takes a key and produces the item's position, and the table's speed depends largely on its quality. A good one spreads keys evenly, so every operation takes a step or two; a bad one piles keys into a few buckets, and the table is no better than a list.

12.2.1 What makes a good hash function

A good hash function has four properties:

- **It is deterministic.** The same key always gives the same index. If "Ali" goes into bucket 8 but is later sought in bucket 3, it is lost. So the function cannot depend on random numbers, the clock, or anything outside the key.
- **It is fast.** A slow hash throws away the table's whole advantage. We treat its cost as *O(1)*: hashing a string takes time proportional to its length, but key length does not grow with the number of items.
- **It distributes keys uniformly.** When buckets fill evenly, each holds few items and fewer keys share a bucket — that is, there are fewer collisions.
- **It uses the whole key.** A function that sees only the first letter sends "Ali", "Ayşe", and "Ahmet" to the same bucket. If some part of the key is ignored, keys that agree elsewhere always collide.

Numbers show why the last point matters. Put "Ali", "Ayşe", and "Ahmet" into 13 buckets. Using only the first letter's code (A = 65) sends all three to bucket *65 mod 13 = 0*; Java's `String.hashCode()`, which uses every character, puts them in buckets 8, 6, and 5. Changing only the function takes us from three collisions to none.

The worst function returns the same index for every key. Then all items share one bucket, a search walks through them in *O(n)*, and the hash table has become a complicated linked list.

The **capacity**, the number of buckets *N*, matters too: too many buckets waste memory, too few crowd up and collide. In practice *N* is kept proportional to the number of items, and the table grows as items arrive (12.5.2).

> **Common mistake**
>
> Hashing a value that can change, such as a counter or the current time. If the hash changes after insertion, a lookup searches the wrong bucket and misses the item. Keys whose fields change after they enter the table cause the same problem (12.2.3).

12.2.2 Hash codes and compression

A hash function works in two stages, shown for "Ali" in Figure 12.5.

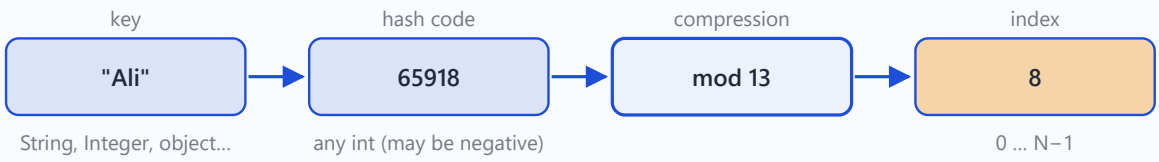


Figure 12.5: The two stages of a hash function. The key ("Ali"; it could be a string, an integer, or any object) first becomes a **hash code**, 65918, which can be any `int`, even a negative one. **Compression** then brings it into the table's range: *65918 mod 13 = 8*, an index from 0 to *N − 1*.

1. **Hash code:** the key becomes an integer, independently of the table. In Java this is each object's `hashCode()`. The code knows nothing of the table's size and can be huge or negative.
2. **Compression function:** the hash code is reduced to an index from 0 to $N - 1$. This step depends on the table's size; if the table grows, the same code compresses to a different index.

The split pays off: the hash code is the key's job (`String` and `Integer` each compute their own), compression is the table's. So one table works with keys of any type, and one key fits a table of any size.

Two compression methods are common.

The division method. $h(k) = k \bmod N$: the remainder of the hash code divided by N, always from 0 to $N - 1$ (for negative codes see 12.3.4). It is simple and fast but sensitive to N: with a prime N, regular patterns in the keys — say, all multiples of 10 — do not pile up in the same buckets.

k	$k \bmod 10$	$k \bmod 11$
10	0	10
20	0	9
30	0	8
40	0	7
50	0	6

Table 12.3: The division method on multiples of 10. With $N = 10$ all five share a bucket; with $N = 11$ (prime) all five are apart.

Why? With $N = 10$, $k \bmod 10$ sees only the last digit, which is 0 for every key, so the other digits have no effect — a breach of the "whole key" rule. 11 shares no factor with 10, so the remainders run 10, 9, 8, … across the buckets. In general, if the keys are multiples of d and g is the greatest common divisor of d and N, the keys can reach only $1/g$ of the buckets. With a prime N, g exceeds 1 only if d is a multiple of N.

The MAD method (Multiply-Add-and-Divide). $h(k) = ((a \cdot k + b) \bmod p) \bmod N$, where p is a prime larger than N, and $a \neq 0$ and b are chosen at random when the table is created. Multiplying and adding scramble patterns in the keys, so the spread is good even when N is not prime.

Example 12.2

Apply MAD to the keys 10, 20, 30, 40, 50 with $N = 10$, $a = 7$, $b = 3$, $p = 101$.

1. **k = 10:** $7 \cdot 10 + 3 = 73$; $73 \bmod 101 = 73$; $73 \bmod 10 = 3$.
2. **k = 20:** $143 \bmod 101 = 42$; $42 \bmod 10 = 2$.
3. **k = 30:** $213 \bmod 101 = 11$; $11 \bmod 10 = 1$.

4. **k = 40:** *283 mod 101 = 81*; *81 mod 10 = 1*.
5. **k = 50:** *353 mod 101 = 50*; *50 mod 10 = 0*.
6. **Result:** buckets 3, 2, 1, 1, 0. The division method with *N = 10* put all five keys in one bucket; MAD uses four, and only 30 and 40 collide. The goal is fewer collisions; eliminating them is usually impossible.

12.2.3 The equals and hashCode contract

A Java hash table relies on two methods: `hashCode()` picks a key's bucket, and `equals()` tells whether a key found there really is the one sought. The two must agree, and Java states the agreement as a **contract**.

Definition 12.3

The `equals` / `hashCode` contract: if `a.equals(b)` is true, then `a.hashCode() == b.hashCode()` must hold. The converse is not required: objects with the same hash code need not be equal. An object's `hashCode()` must also stay the same while the fields used by `equals` are unchanged.

Four consequences follow:

- **Equal objects** have the same hash code. Otherwise equal keys land in different buckets, and a value inserted under `a` cannot be found under `b`.
- **Equal hash codes** do not imply equal objects: "Aa" and "BB" both hash to 2112. That is why the table, having reached a bucket, still compares keys with `equals`.
- **Override both.** A class that redefines (*overrides*) `equals` must override `hashCode` on the same fields; overriding only one breaks the contract.
- **Mutable keys.** If a key's `equals` / `hashCode` fields change after insertion, its new hash code points to another bucket while the item stays in the old one.

Java 16's `record` types get both methods right automatically:

```
record Student(int id, String name) {}
Map<Student, Double> grades = new HashMap<>();
grades.put(new Student(42, "Ada"), 87.5); // record: equals/hashCode built in (Java 16+)
```

`record Student(int id, String name) {}` defines an immutable class whose generated `equals` compares both fields and whose `hashCode` is computed from both. A grade inserted under `new Student(42, "Ada")` is found with a different `new Student(42, "Ada")` object, and since record fields cannot change, the mutable-key problem cannot arise.

12.2.4 Forgetting hashCode

The contract is most often broken by overriding `equals` and forgetting `hashCode`. The `Point` class below treats points with equal coordinates as equal but has no `hashCode()`.

```
class Point {
  int x, y;
  Point(int x, int y) { this.x = x; this.y = y; }
  public boolean equals(Object o) {
    return o instanceof Point p
        && x == p.x && y == p.y;
  }
  // no hashCode() -> Object's is used
}
Map<Point, String> m = new HashMap<>();
m.put(new Point(1, 2), "A");
m.get(new Point(1, 2));  // most likely null (!)
```

`o instanceof Point p` asks whether `o` is a `Point` and, if so, binds it to `p`; then the coordinates are compared. By this method, two `new Point(1, 2)` objects are equal.

Without an override, Java uses `Object`'s `hashCode()`, which is based on the object's *identity* and usually differs for every new object. So two points with the same coordinates almost always have different identity hashes, with the result shown in Figure 12.6.

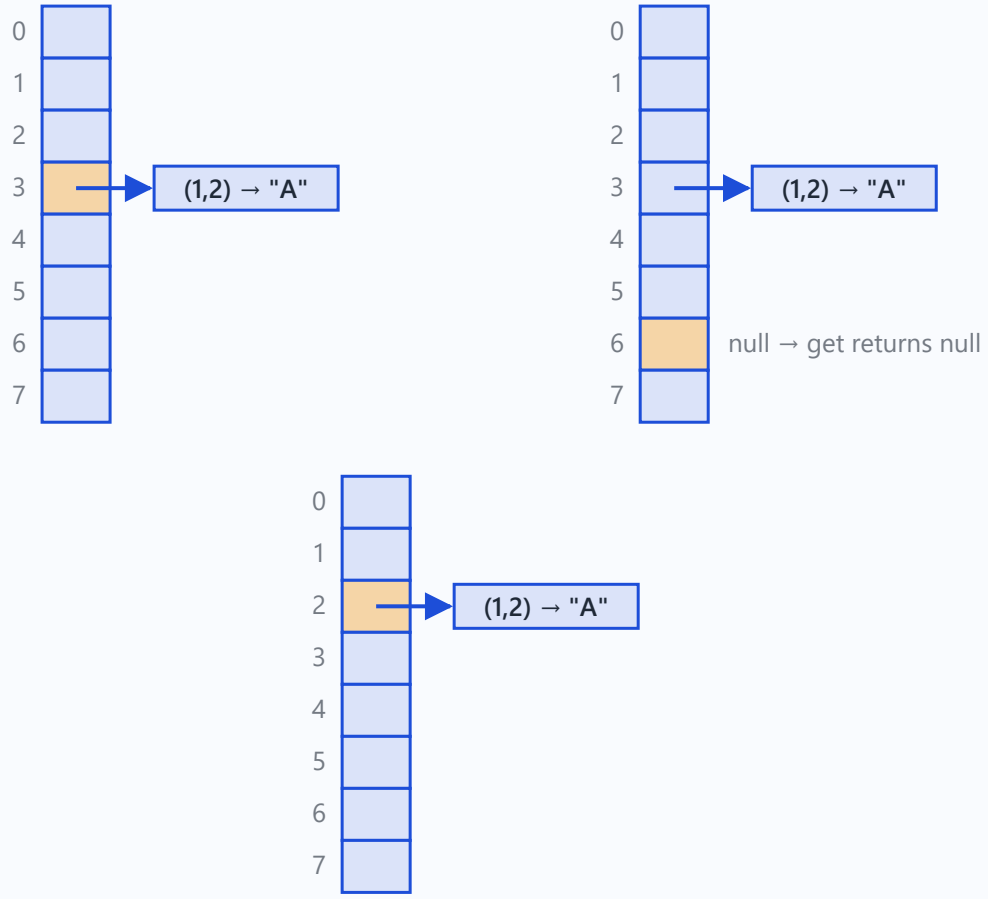


Figure 12.6: When `hashCode` is forgotten. (a) `put(p1, "A")`: the bucket comes from the identity-based `p1.hashCode()`, say bucket 3. (b) `get(p2)`: `p2` equals `p1` but has a different identity hash, say bucket 6, which is empty, so the result is `null`; `equals` is never called. (c) The fix, `return Objects.hash(x, y);`: equal points share a bucket (`Objects.hash(1, 2) = 994`, and *994 mod 8 = 2* with 8 buckets), where `equals` finds the match.

Buckets 3 and 6 are only examples; identity hashes can vary from run to run. The outcome almost never does: `get` looks in the wrong bucket and returns `null`. "Almost" matters — if two identity hashes happen to share a bucket, `get` succeeds and the bug hides. Bugs that make a program fail only sometimes are among the hardest to find.

Moreover, `HashMap` compares hash codes before calling `equals` on a node, so even when two such points share a bucket, differing identity hashes mean `equals` is never called. Without a `hashCode` override, however correct `equals` is, the map never sees the equality.

The fix is to override `hashCode` on the fields `equals` uses:

```
public int hashCode() {
  return Objects.hash(x, y);
}
```

`Objects.hash(x, y)` combines the fields much as `String.hashCode()` does: $31 \cdot (31 \cdot 1 + x) + y$, which is $31 \cdot 32 + 2 = 994$ for (1, 2). Equal points give equal values and share a bucket, so the contract holds. `Point(2, 1)` gives $31 \cdot 33 + 1 = 1024$: order matters.

> **Common mistake**
>
> Computing `hashCode` from a field `equals` ignores. If `equals` compares only student IDs but `hashCode` uses the ID and the name, two records with the same ID and differently spelled names are equal yet hash differently, breaking the contract. `hashCode` may use only fields that `equals` looks at.

12.2.5 Some hash functions

Here are four hash functions in common use, each with a short Java version.

Modulus (division remainder). The simplest function for integer keys:

```
int modulusHash(int key, int arraySize) {
  return Math.floorMod(key, arraySize);
}
```

Java's `%` takes the sign of the dividend, so a negative key can give a negative remainder; `Math.floorMod` always returns 0 … `arraySize - 1` for a positive divisor (12.3.4).

CRC32. The cyclic redundancy check is a checksum designed to detect corrupted data in transmission; it mixes bits well enough to serve as a hash function. Java provides it in `java.util.zip`:

```
long crc32Hash(String key) {
  CRC32 crc = new CRC32();  // java.util.zip
  crc.update(key.getBytes());
  return crc.getValue();
}
```

The method feeds the string's bytes to a `CRC32` object and returns the 32-bit result, from 0 to $2^{32} - 1$, as a `long`; it still needs compressing. For "Ali" it returns 1933558706.

DJB2. A short, fast string hash by Daniel J. Bernstein:

```
int djb2Hash(String s) {
  int h = 5381;
  for (int i = 0; i < s.length(); i++)
    h = (h * 33) ^ s.charAt(i);
  return h;
}
```

Starting at 5381, each character multiplies the result by 33 and combines it with the character's code. Classic DJB2 adds (`h * 33 + c`); this common variant uses XOR (`^`, bitwise exclusive or). The multiplication soon overflows `int`, so results may be negative: "Ali" gives 193442657, but long strings often go negative, and a safe compression is again needed.

Java's `String.hashCode()`. The function Java uses for strings:

```
// Σ s[i] · 31^(n-1-i)
"abc".hashCode()  // 96354
```

For n characters it computes $s[0] \cdot 31^{n-1} + s[1] \cdot 31^{n-2} + \ldots + s[n-1]$, weighting each character by a power of 31 that depends on its position. For "abc", $97 \cdot 31^2 + 98 \cdot 31 + 99 = 93217 + 3038 + 99 = 96354$. Thanks to the positional weights, "abc" and "cba" differ; simply adding the codes could not tell them apart.

Why 31? It is odd and prime. An odd multiplier does not wipe out the information in overflowing bits, whereas an even one would zero the lowest bit at every step. And $31 \cdot h$ is fast to compute as $(h << 5) - h$, a shift and a subtraction.

Common mistake

Assuming different strings get different hash codes. "Aa" and "BB" both give 2112 ($65 \cdot 31 + 97 = 2112$ and $66 \cdot 31 + 66 = 2112$), and so "AaAa", "AaBB", and "BBBB" all give 2031744. A hash code is not an ID; equality is always checked with `equals`.

12.2.6 Code trace: "Ali".hashCode()

The `hash` method below does `String.hashCode()`'s calculation with an explicit loop; `main` applies it to "Ali" and compresses the result for 13 buckets.

```
// same calculation as String.hashCode():
int hash(String s) {
  int h = 0;
  for (int i = 0; i < s.length(); i++)
    h = 31 * h + s.charAt(i);
  return h;
}

// main:
int code = hash("Ali");               // 65918
int index = Math.floorMod(code, 13);  // 8
```

Starting from `h = 0`, each iteration multiplies `h` by 31 and adds the next character's code; Java converts the `char` from `s.charAt(i)` to its Unicode code, an integer, in the addition. In "Ali", A = 65, l = 108, i = 105. Table 12.4 lists each iteration.

`i`	**Condition** `i < 3`	`s.charAt(i)`	**Calculation**	`h` **(after)**
—	—	—	start	0
0	true	'A' (65)	*31 · 0 + 65*	65
1	true	'l' (108)	*31 · 65 + 108*	2123
2	true	'i' (105)	*31 · 2123 + 105*	65918
3	false	—	loop ends, `return`	65918

Table 12.4: Trace of `hash("Ali")`.

The formula agrees: $65 \cdot 31^2 + 108 \cdot 31 + 105 = 62465 + 3348 + 105 = 65918$. The loop never computes powers; multiplying the running total by 31 raises every earlier character's coefficient by one power. This is **Horner's method** for evaluating polynomials: n multiplications and n additions, $O(n)$ time, $O(1)$ extra space.

Finally, `Math.floorMod(65918, 13)`: $13 \cdot 5070 = 65910$ and $65918 - 65910 = 8$, so "Ali" lands in bucket 8 (Figure 12.7).

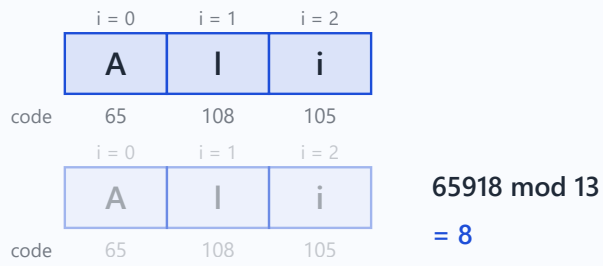


Figure 12.7: Tracing "Ali".hashCode(). Top: the characters and their Unicode codes (A = 65, l = 108, i = 105). Bottom: the loop has ended with `code = 65918`, and *65918 mod 13 = 8*.

Short strings give positive results, but long ones exceed the `int` limit ($2^{31} - 1 = 2{,}147{,}483{,}647$). Java raises no error on overflow; the value silently wraps and may turn negative — "Sercan".hashCode() is −1821977712. Stranger still, "polygenelubricants" hashes to exactly `Integer.MIN_VALUE`, −2147483648, a value that exposes a trap in 12.3.4.

12.2.7 Code trace: modulusHash()

Here is the division method on a 10-cell table: each of five keys gets a cell index and is written there.

```
int modulusHash(int key, int arraySize) {
  return Math.floorMod(key, arraySize);
}

// main:
Integer[] table = new Integer[10];
for (int k : new int[]{5, 1, 10, 26, 99}) {
  int j = modulusHash(k, table.length);
  table[j] = k;
}
```

Because `table` is an `Integer` array rather than `int`, every cell starts as `null`, keeping an empty cell distinct from 0. Each iteration writes the key into cell `k mod 10`.

`k`	`k mod 10`	**Cell written**	`table` **(after)**
—	—	—	[·, ·, ·, ·, ·, ·, ·, ·, ·, ·]
5	5	`table[5]`	[·, ·, ·, ·, ·, 5, ·, ·, ·, ·]
1	1	`table[1]`	[·, 1, ·, ·, ·, 5, ·, ·, ·, ·]
10	0	`table[0]`	[10, 1, ·, ·, ·, 5, ·, ·, ·, ·]
26	6	`table[6]`	[10, 1, ·, ·, ·, 5, 26, ·, ·, ·]
99	9	`table[9]`	[10, 1, ·, ·, ·, 5, 26, ·, ·, 99]

Table 12.5: Trace of `modulusHash`, cells 0 to 9; "·" marks an empty cell.

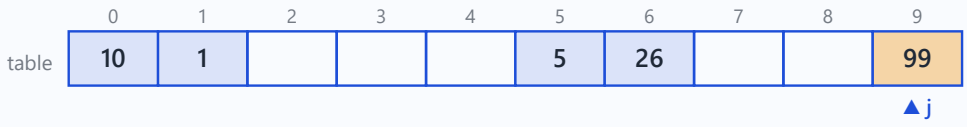


Figure 12.8: End of the `modulusHash` trace: keys 5, 1, 10, 26, and 99 sit in five different cells, 5, 1, 0, 6, and 9.

No cell got two keys, but only by luck: a next key of 15 would go to cell *15 mod 10 = 5* and overwrite 5. This table ignores collisions and is for illustration only. Each insertion took *O(1)*, the five together *O(5)*, in a fixed space of 10 cells.

12.2.8 Collisions

Definition 12.4

A **collision** occurs when two different keys $k_1 \neq k_2$ have $h(k_1) = h(k_2)$.

With 10 buckets and the division method, for example, $h(5) = h(105) = 5$. Two items cannot share a cell directly, so the table needs a strategy for this case.

Collisions are no accident: whenever the key space is larger than the table, a simple counting argument makes them unavoidable.

Theorem 12.1 (Pigeonhole principle)

Any hash function that sends more than N distinct keys into N buckets produces at least one collision. More generally, with U possible keys, some bucket receives at least $\lceil U / N \rceil$ of them.

Proof. Each of $N + 1$ keys goes to one of N buckets. If every bucket held at most one key, there would be at most N keys; so some bucket holds two. Likewise, if every bucket received fewer than $\lceil U / N \rceil$ keys, the total would fall short of U. □

This limits what any hash function can do. For nine-digit IDs, $U = 10^9$, and with 1000 buckets the fullest bucket receives at least a million possible IDs. A good function cannot prevent collisions; it can only spread the keys that *actually arrive* evenly. And collisions come early: 23 random keys in 365 buckets collide with probability about 0.507. This is the "birthday problem" — in a class of 23, two people probably share a birthday.

In Figure 12.9, six key-value pairs go into a 10-slot table by the division method. The first five find empty slots, but (105, Murat) wants slot 5, which holds (5, Ali).

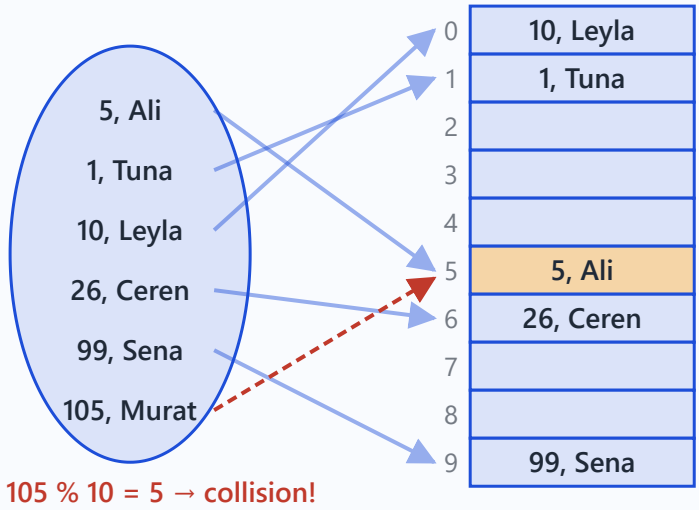


Figure 12.9: A collision. Pairs go into 10 slots by *k mod 10*: 5 → 5, 1 → 1, 10 → 0, 26 → 6, 99 → 9. (105, Murat) also wants slot 5, where (5, Ali) sits: *105 % 10 = 5*.

There are two main strategies for resolving collisions:

- **Separate chaining:** each bucket keeps a linked list of the items that land in it, and a colliding item joins the list.
- **Open addressing:** all items live in the array itself, and an item arriving at an occupied slot looks for another empty one by a fixed rule.

The next two sections code and trace each strategy.

12.3 Separate chaining

Separate chaining resolves collisions in the most direct way: a bucket holds every item that lands in it. Instead of one item, each bucket stores the head of a linked list called a **chain**; "separate" means each bucket has its own.

12.3.1 The idea of chaining

Insert (5, "Ali"), (1, "Tuna"), (10, "Leyla"), (26, "Ceren"), (99, "Sena"), and (105, "Murat"), in that order, into 10 buckets with *h(k) = k % 10*:

- *h(5) = 5*: bucket 5 is empty; the node (5, "Ali") is attached to it.
- *h(1) = 1*: bucket 1 is empty; the node (1, "Tuna") is attached to it.
- *h(10) = 0*: bucket 0 is empty; the node (10, "Leyla") is attached to it.
- *h(26) = 6*: bucket 6 is empty; the node (26, "Ceren") is attached to it.
- *h(99) = 9*: bucket 9 is empty; the node (99, "Sena") is attached to it.

- *h(105) = 5*: bucket 5 already holds (5, "Ali") — a collision. The new node goes at the **front**: the bucket points to 105, and 105 to 5.

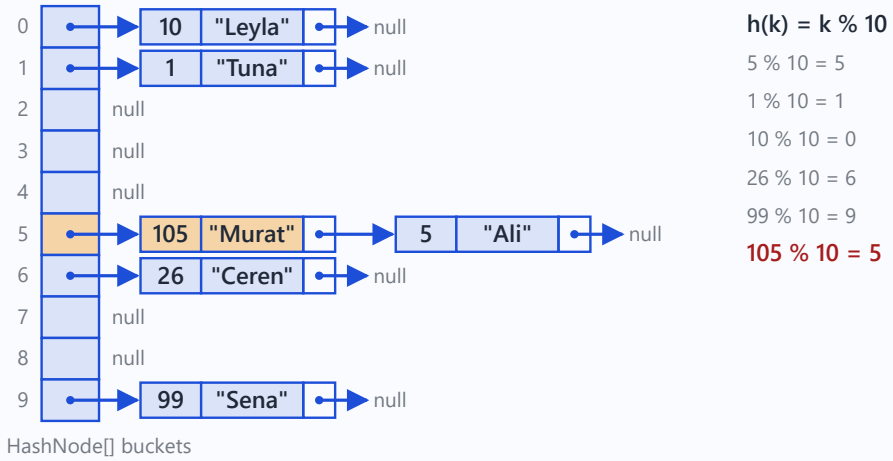


Figure 12.10: The chained table after six pairs. Each cell of `HashNode[] buckets` holds a chain's first node or `null`. 105 and 5 share bucket 5 (*105 % 10 = 5*); 105 came later and heads the chain: bucket 5 → (105, "Murat") → (5, "Ali") → `null`.

Why the front? The bucket knows only the first node, so appending means walking to the end, while prepending takes two reference assignments — new node to old head, bucket to new node — in *O(1)* regardless of length. But as 12.3.5 shows, insertion first checks whether the key is already in the chain, which walks the whole chain anyway; so some implementations, Java's `HashMap` among them, append. Both are correct.

A chained table never fills up; chains just grow. That flexibility is the method's strength, and its price is that longer chains mean slower searches, so we watch the average chain length — how full the table is.

12.3.2 Hash table terminology

To ease reading the code, Figure 12.11 names the parts of the same six-pair table.

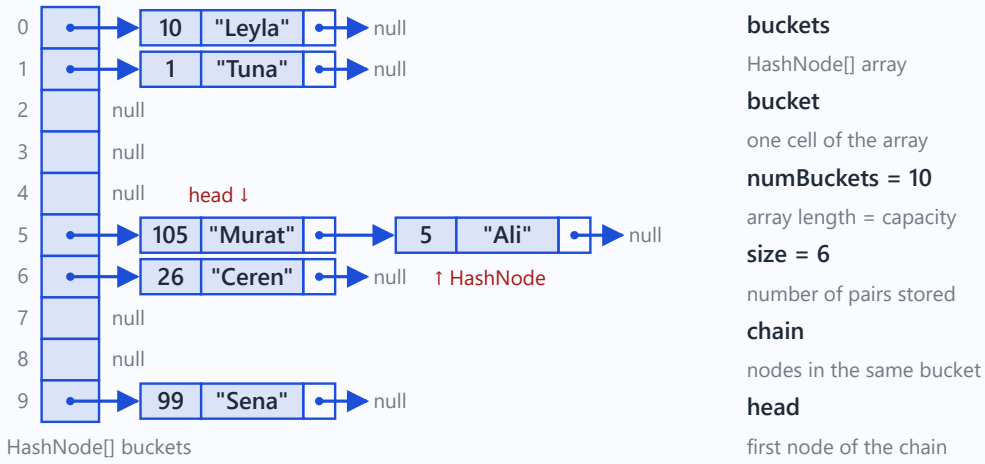


Figure 12.11: Parts of a chained hash table: `buckets` (the `HashNode[]` array), a **bucket** (one cell), `numBuckets = 10` (array length, the capacity), `size = 6` (pairs stored), a **chain** (the nodes in one bucket), `head` (a chain's first node), and `HashNode` (key, value, and next reference).

Term	Name in code	Meaning	In the figure
Bucket array	`buckets`	Array of chain heads.	10 cells
Bucket	`buckets[i]`	One cell; `null` if empty.	0 … 9
Capacity	`numBuckets`	Number of buckets, *N*.	10
Size	`size`	Number of pairs stored, *n*.	6
Chain	—	List of nodes in one bucket.	bucket 5: 105 → 5
Node	`HashNode`	Key, value, and next reference.	(26, "Ceren")

Table 12.6: Parts of a chained hash table and their names in the code.

Capacity counts buckets; size counts pairs. In the figure, 5 of 10 buckets are occupied but there are 6 pairs, since one bucket holds two. Under chaining, size can exceed capacity: 10 buckets can hold 25 pairs.

12.3.3 Code: the HashTable class

Here is the skeleton of a chained hash table with `Integer` keys and `String` values.

```
public class HashTable {
  private HashNode[] buckets;      // first nodes of the chains
  private int numBuckets, size;

  public HashTable(int capacity) {
    this.numBuckets = capacity;
    this.buckets = new HashNode[capacity];
    this.size = 0;
  }
  private class HashNode {
    private Integer key;
    private String value;
    private HashNode next;
    public HashNode(Integer key, String value) {
      this.key = key; this.value = value;
    }
  }
}
```

- `buckets` is an array of `HashNode`; each cell points to a chain's first node, or is `null`.
- `numBuckets` is the array's length, `size` the number of pairs.
- The constructor allocates the array. A new Java object array starts out all `null`, so no loop is needed: `new HashTable(10)` gives 10 empty buckets and `size = 0`.
- The inner class `HashNode` holds a `key`, a `value`, and a `next` reference — a singly linked list node with a key added.

The node must store the key, not just the value: a bucket can hold several pairs, and only the stored key tells which node is wanted. With values alone, we could not tell whether "Murat" or "Ali" in bucket 5 belongs to 105.

Space: *N* references in the array plus a key, a value, and a `next` per node, *O(N + n)* in all.

12.3.4 Bucket indices and negative hash codes

`getBucketIndex` computes a key's bucket:

```
// Naive way: key % buckets.length → can be negative!
private int getBucketIndex(Integer key) {
  int h = key.hashCode();  // = key
  return Math.floorMod(h, buckets.length);
}
```

An `Integer`'s `hashCode()` is the number itself (41 hashes to 41), which `Math.floorMod` compresses to the array length. The first line's comment is a warning: the naive `key % buckets.length` can be negative.

Java's `%` takes **the sign of the dividend**: *41 % 10 = 1*, but *−7 % 10 = −7*, and a negative index throws `ArrayIndexOutOfBoundsException`. `Math.floorMod` always returns 0 to divisor − 1 for a positive divisor.

`key`	`key % 10`	`Math.floorMod(key, 10)`	**Result**
105	5	5	bucket 5
21	1	1	bucket 1
41	1	1	bucket 1 (collides with 21)
−7	−7	3	with `%`, `buckets[-7]` → `ArrayIndexOutOfBoundsException`

Table 12.7: `%` versus `Math.floorMod` with 10 buckets.

This matters even if keys are always positive integers: a table meant for any key type must handle keys like `String`s, whose hash codes are often negative ("Sercan" hashes to −1821977712).

For a divisor *n > 0*, `Math.floorMod` behaves like this short method:

```
private int getBucketIndex(Integer key) {
  int h = key.hashCode();  // = key
  return Math.floorMod(h, buckets.length);
}

// equivalent of floorMod for n > 0:
static int floorMod(int x, int n) {
  int r = x % n;       // sign comes from x
  if (r < 0) r += n;   // if negative, add n
  return r;
}
```

Since *x % n* lies between *−(n − 1)* and *n − 1*, adding *n* to a negative result moves it into 1 … *n − 1*; positive results are untouched. Table 12.8 traces two calls.

Step	`key = 41`	`key = -7`
`h = key.hashCode()`	41	−7
`r = h % 10`	1	−7 (sign of the dividend)
`r < 0`?	no correction	yes: `r += 10` → 3
Returns	1 (collides with 21)	3

Table 12.8: Trace of `getBucketIndex(41)` and `getBucketIndex(-7)` (10 buckets).

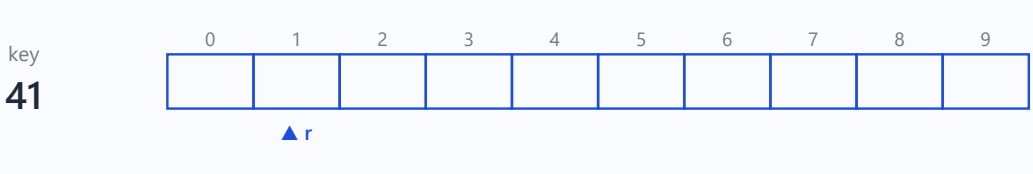




Figure 12.12: `getBucketIndex` results. Top: 41 gives `r = 1`, bucket 1. Bottom: for −7, `%` gives −7, and `r += 10` makes it 3, bucket 3.

Taking the absolute value, `Math.abs(h) % N`, almost always works, with one exception. An `int` spans −2147483648 to 2147483647, so the negation of `Integer.MIN_VALUE = -2147483648` does not fit, and `Math.abs(Integer.MIN_VALUE)` overflows back to −2147483648.

Example 12.3

"polygenelubricants", whose hash code is −2147483648, is inserted into 13 buckets. Compare three index calculations.

1. `h % 13`: −2147483648 % 13 = −11, a negative index: `ArrayIndexOutOfBoundsException`.
2. `Math.abs(h) % 13`: `Math.abs(h)` is still −2147483648, so the result is again −11. The absolute value does nothing.
3. `Math.floorMod(h, 13)`: −11 + 13 = 2; the key goes safely into bucket 2.
4. **Result:** this crash appears only when an input happens to hit one rare value, which is why index calculations always use `Math.floorMod(h, N)`.

12.3.5 Code: put()

Insertion has three steps: find the bucket, scan the chain, and if needed add at the front.

```
public void put(Integer key,
                String value) {
  int bucketIndex = getBucketIndex(key);
  HashNode head = buckets[bucketIndex];
  while (head != null) {    // is the key there?
    if (head.key.equals(key)) {
      head.value = value;  // if so, update
      return;
    }
    head = head.next;
  }
  size++;
  HashNode node =
      new HashNode(key, value);
  node.next = buckets[bucketIndex];
  buckets[bucketIndex] = node;   // add at the front
}
```

- `getBucketIndex(key)` finds the bucket in *O(1)*; no other bucket is ever examined.
- `head = buckets[bucketIndex]` starts at the chain's first node and, despite its name, moves along the chain in the loop.
- The `while` loop asks `head.key.equals(key)` at each node. If the key is found, only its value changes and the method returns; this is an update, so `size` stays the same.
- If the loop ends with `head == null`, the key is absent: `size` grows by one and a new node is created.
- `node.next = buckets[bucketIndex]` links the node in front of the old chain, and `buckets[bucketIndex] = node` points the bucket to it. The order matters.

Keys are compared with `equals`, not `==`, which asks whether two references point to the same object. Java caches `Integer` objects for −128 … 127, so `==` seems to work there; from 128 up, equal values are usually different objects and `==` returns `false`.

Cost. Finding the bucket and adding at the front are *O(1)*, and the scan is proportional to chain length: *O(1 + chain length)*, constant on average while chains stay short (12.3.10).

> **Common mistake**
>
> Swapping the two lines of the front insertion. If `buckets[bucketIndex] = node` comes first, `node.next = buckets[bucketIndex]` then links the node to itself, and the whole old chain is silently lost. Link the new node to the old chain first, then update the bucket.

12.3.6 Code trace: put()

A 10-bucket table holds only (105, "Murat"), so `size = 1`. We call `put(21, "Leyla")`, `put(41, "Sena")`, and `put(21, "Ali")` (Table 12.9).

Call	Bucket	Chain scan	Result	Bucket 1 (after)	`size`
`put(21, "Leyla")`	*21 mod 10 = 1*	empty, `head = null`; loop skipped	new node at front	21 → `null`	2
`put(41, "Sena")`	*41 mod 10 = 1*	21 ≠ 41; `head = null`	new node at front	41 → 21 → `null`	3
`put(21, "Ali")`	*21 mod 10 = 1*	41 ≠ 21; 21 = 21	value updated: "Leyla" → "Ali"	41 → 21 → `null`	3

Table 12.9: Trace of three `put` calls. "Bucket 1" shows the chain front to back.

In the first call bucket 1 is empty, so the loop never runs and the key is known to be absent. `size` becomes 2, and the node (21, "Leyla") gets `next = buckets[1]`, which is `null`, before the bucket is pointed at it.

In the second, 41 also lands in bucket 1: a collision. The loop sees *21 ≠ 41*, advances, and reaches `null`. `size` becomes 3, and the node (41, "Sena") points to the old head 21 while the bucket points to 41. The chain is now 41 → 21; the old chain sits behind the new node.

The third call is an update. The loop passes 41, then finds `21.equals(21)` true, changes the value from "Leyla" to "Ali", and returns, after two comparisons. No node is added and `size` stays 3.

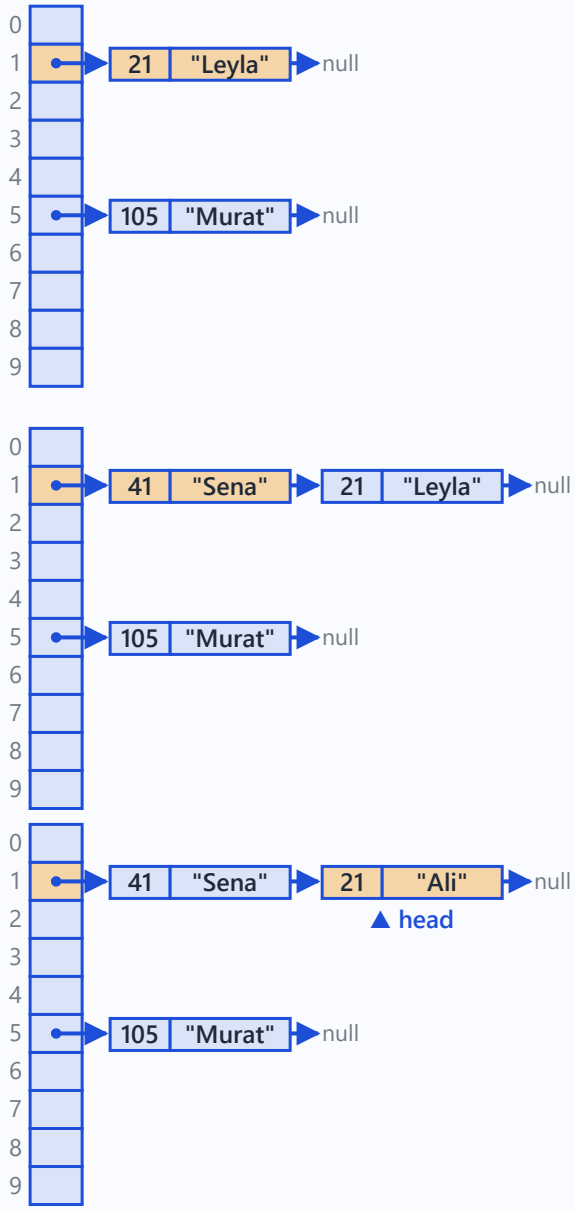


Figure 12.13: The `put` trace. (a) After `put(21, "Leyla")`: bucket 1 → 21. (b) After `put(41, "Sena")`: the new node heads the chain, bucket 1 → 41 → 21. (c) `put(21, "Ali")`: 21 is found and its value becomes "Ali"; no node is added.

12.3.7 Code and trace: get()

Lookup is the scanning part of `put`:

```
public String get(Integer key) {
  int bucketIndex = getBucketIndex(key);
  HashNode head = buckets[bucketIndex];
  while (head != null) {
    if (head.key.equals(key))
      return head.value;       // found
    head = head.next;
  }
  return null;                 // no such key
}
```

The method scans only the key's own chain, returning the value if the key is found and `null` if the chain runs out. The cost is *O(1 + chain length)*, and the average chain length is *n / N*. Three searches on the table from the previous trace (bucket 5 → 105; bucket 1 → 41 → 21):

Call	Bucket	Nodes examined	Comparisons	Returns
`get(105)`	*105 % 10 = 5*	105	1	"Murat"
`get(21)`	*21 % 10 = 1*	41, 21	2	"Ali"
`get(88)`	*88 % 10 = 8*	— (empty)	0	`null`

Table 12.10: Three `get` calls.

`get(105)` matches at the first node. `get(21)` first meets 41, a different key in the same bucket — the trace of a collision — and finds 21 at the second node. `get(88)` finds an empty bucket and returns `null` in one step. Had that bucket been occupied, the whole chain would have been scanned, since absence becomes certain only at the chain's end.

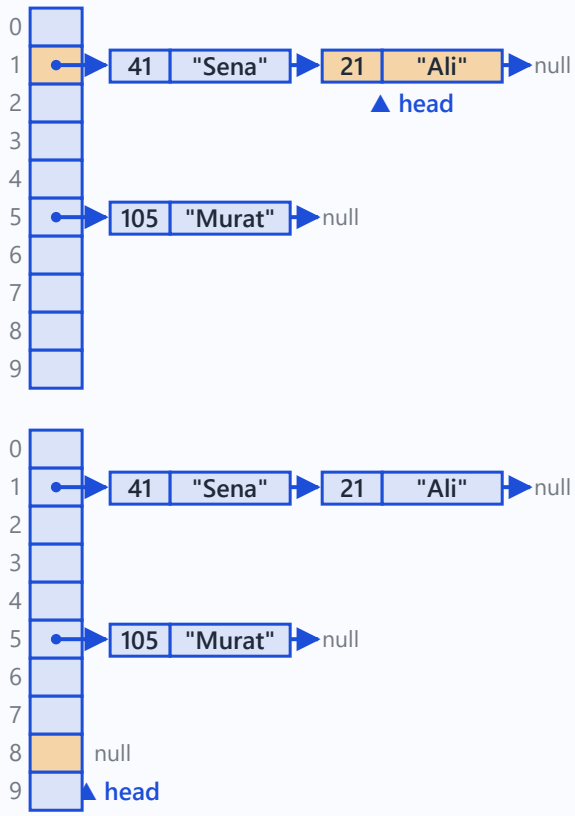


Figure 12.14: Tracing `get`. (a) `get(21)`: `head` reaches the second node, 21 = 21, and "Ali" is returned after 2 comparisons. (b) `get(88)`: bucket 8 is empty, `head = null`, and `null` is returned.

12.3.8 Code: remove()

Removing a node from a chain is deletion from a singly linked list: we must change the `next` field of the node *before* it, so the scan tracks the previous node too.

```
public String remove(Integer key) {
  int bucketIndex = getBucketIndex(key);
  HashNode head = buckets[bucketIndex];
  HashNode prev = null;
  while (head != null &&
         !head.key.equals(key)) {
    prev = head;
    head = head.next;
  }
  if (head == null) return null;   // absent
  size--;
  if (prev != null)         // middle or end
    prev.next = head.next;
  else                      // front
    buckets[bucketIndex] = head.next;
  return head.value;
}
```

- `prev` starts as `null`: nothing precedes the first node.
- The loop advances while the chain continues *and* the key does not match. Java's `&&` skips its second test when the first is false, so `head.key` is never read when `head == null` and no `NullPointerException` occurs.
- Each iteration sets `prev = head` and `head = head.next`, moving both references together.
- After the loop, `head == null` means the key is absent and the method returns `null`; otherwise `head` is the node to delete.
- In the middle or at the end (`prev != null`), the previous node skips it: `prev.next = head.next`. At the front (`prev == null`), the bucket itself changes: `buckets[bucketIndex] = head.next`.
- `size` has been decremented and the removed value is returned. Nothing references the unlinked node any more, so the garbage collector reclaims it.

The cost equals `get`'s, *O(1 + chain length)*; unlinking is a few assignments.

> **Common mistake**
>
> Deleting the front node with only `prev.next = head.next`. At the front, `prev` is `null` and this line throws a `NullPointerException`. Front deletion must update the bucket itself, which is why the code splits the cases with `if` / `else`.

12.3.9 Code trace: remove()

Start again from bucket 5 → (105, "Murat") and bucket 1 → (41, "Sena") → (21, "Ali"), `size = 3`, and call `remove(21)`, `remove(105)`, and `remove(88)`.

Call	Loop steps	After the loop	Branch	Table (after)	`size`	Returns
`remove(21)`	41 ≠ 21 → `prev = 41`, `head = 21`; 21 = 21, loop ends	`prev` = 41, `head` = 21	`prev.next = head.next` (`null`)	1 → 41; 5 → 105	2	"Ali"
`remove(105)`	105 = 105, loop skipped	`prev = null`, `head` = 105	`buckets[5] = head.next` (`null`)	1 → 41	1	"Murat"
`remove(88)`	bucket 8 empty, `head == null`	`head = null`	`return null`	unchanged	1	`null`

Table 12.11: Trace of three `remove` calls.

`remove(21)` deletes from the middle. No match at 41, so both references advance: `prev` to 41, `head` to 21. 21 matches and the loop ends; since `prev` is not `null`, `prev.next = head.next` makes 41 skip 21 and point to `null`.

`remove(105)` deletes from the front. The condition fails at the first node, so the body never runs and `prev` stays `null`; the `else` branch sets `buckets[5] = head.next`, and with nothing after 105 the bucket empties.

`remove(88)` fails: the bucket is empty, the loop ends at once, and `null` is returned with the table and `size` unchanged.

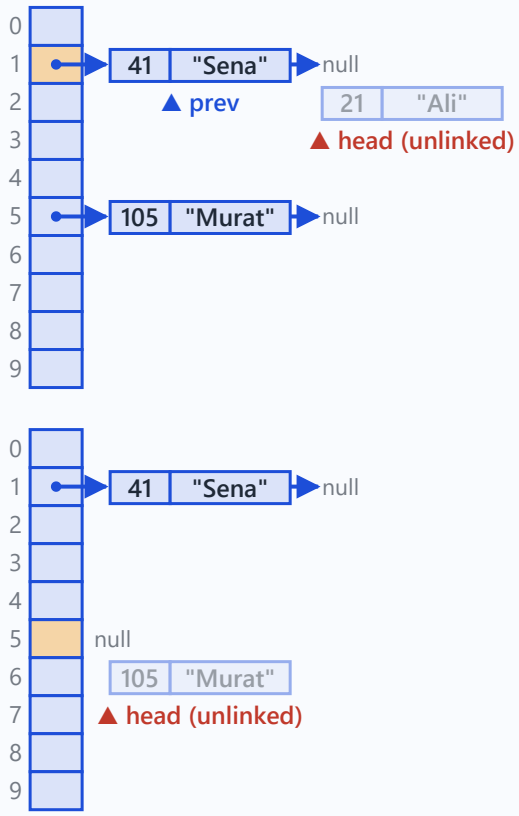


Figure 12.15: Tracing `remove`. (a) `remove(21)`: `prev.next = head.next`; 41 now skips 21, which is unlinked. (b) `remove(105)`: the node headed the chain, and `buckets[5] = head.next` empties the bucket.

12.3.10 The cost of chaining

Every operation on a chained table takes $O(1 + \text{chain length})$. How long a chain gets depends on how full the table is.

Definition 12.5

The **load factor** of a hash table with N buckets and n pairs is $\lambda = n / N$. Under chaining, λ is the average chain length.

An average is not a guarantee: one bucket may be empty and another crowded. The analysis usually assumes that the hash function sends each key to any bucket with equal probability, independently of the other keys — the assumption of **simple uniform hashing**.

Theorem 12.2 (Expected search cost under chaining)

Under simple uniform hashing, in a chained table with load factor λ, an unsuccessful search examines λ nodes on average and a successful one about $1 + \lambda/2$. With the bucket computation, both cost $O(1 + \lambda)$ in expectation.

Proof. An unsuccessful search goes to one of the N buckets with equal probability and scans its whole chain; the chain lengths sum to n, so the expected length is $n / N = \lambda$. In a successful search the key sits behind the keys inserted after it into its chain (new nodes go at the front). Each such key shares its bucket with probability $1/N$, and on average about $(n - 1)/2$ keys come after it. So about $(n - 1)/(2N) \approx \lambda/2$ nodes precede it, and $1 + \lambda/2$ are examined counting the key itself. □

Example 12.4

Compare expected and actual values for the six-pair table of Figure 12.10 ($N = 10$, $n = 6$).

1. **Load factor:** $\lambda = 6 / 10 = 0.6$.
2. **Unsuccessful search:** a random bucket's chain is scanned in full. The chain lengths from bucket 0 to 9 are 1, 1, 0, 0, 0, 2, 1, 0, 0, 1, averaging $6 / 10 = 0.6$ — exactly λ.
3. **Successful search:** searching once for each key, 10, 1, 26, 99, and 105 take one comparison and 5 takes two: 7 in all, an average of $7 / 6 \approx 1.17$ against the theorem's $1 + 0.6/2 = 1.3$. This small table is even better balanced than the assumption predicts.

The theorem's message: keep λ below a constant — say 0.75 — and every operation costs $O(1)$ in expectation. This is done by enlarging the table as pairs arrive, called resizing (12.5.2). Under chaining λ may exceed 1; chains just get longer and operations slower.

The worst case is different. If every key lands in one bucket, a single chain of length n makes search $O(n)$. This happens with a bad hash function or with keys chosen to collide; simple uniform hashing guarantees only the average.

12.4 Open addressing

Separate chaining creates a node per collision, scattered through memory and each carrying an extra reference. Open addressing needs none of this: all pairs live in the array itself, at most one per slot, and a colliding key looks for another empty slot.

12.4.1 Probe sequences

A key first tries its **home slot**, $h(k)$. If it is empty, the key goes there; if not, a fixed rule picks a second slot, then a third, and so on. Checking a slot is called **probing**, and the order of slots tried is the key's **probe sequence**.

Definition 12.6

Open addressing resolves collisions by storing all pairs inside an array of N slots. The ith slot tried for key k is

$$h(k, i) = (h(k) + f(i)) \bmod N, \; i = 0, 1, 2, \ldots$$

Since $f(0) = 0$, the first try is the home slot; the choice of f defines the probing method.

With at most one pair per slot, $n \le N$, so $\lambda \le 1$ — unlike chaining, λ cannot exceed 1. There are three common probing methods:

Method	$f(i)$	Probe sequence	Weakness
Linear probing	i	5, 6, 7, 8, 9, …	Primary clustering
Quadratic probing	i^2	5, 6, 9, 14 → 1, 21 → 8, …	Secondary clustering; may miss slots
Double hashing	$i \cdot h_2(k)$	5, 5 + d, 5 + 2d, … ($d = h_2(k)$)	Needs $h_2(k) \neq 0$ and a step coprime with N

Table 12.12: Three probing methods; sample sequences for $h(k) = 5$, $N = 13$.

Open addressing needs no extra nodes or references, only the array, and a contiguous array suits the processor cache: probing consecutive slots beats hopping between scattered nodes in practice. In return, the load factor must stay low, since probe sequences lengthen quickly as the table fills, and deletion needs care (12.4.5).

All examples in this section insert the keys 18, 41, 22, 44, 59, 32, 31, and 73, in that order, into *N = 13* slots with *h(k) = k mod 13*. Their home slots are 5, 2, 9, 5, 7, 6, 5, and 8. Running the same data through all three methods makes the differences plain.

12.4.2 Linear probing

Linear probing is the simplest method: if the home slot is taken, try the next, then the next, wrapping around at the end:

```
// put(k): linear probing, N = 13
int h = k % 13;                // home slot
for (int i = 0; i < 13; i++) {
  int j = (h + i) % 13;          // h, h+1, h+2…
  if (table[j] == null) {      // empty?
    table[j] = k;
    break;
  }                            // occupied: i++
}
```

`h` is the home slot and `i` counts tries; `j = (h + i) % 13` is the home slot, then the next, and so on, with `% 13` wrapping from 12 to 0. At the first empty slot the key is written and `break` ends the loop, which runs at most 13 times — once per slot.

The first three keys go in on the first try: 18 → 5, 41 → 2, 22 → 9. Table 12.13 traces the rest.

k	Home slot *h*	Slots tried	Final slot	Probes
18	5	5	5	1
41	2	2	2	1
22	9	9	9	1
44	5	5 (18), 6	6	2
59	7	7	7	1
32	6	6 (44), 7 (59), 8	8	3
31	5	5 (18), 6 (44), 7 (59), 8 (32), 9 (22), 10	10	6
73	8	8 (32), 9 (22), 10 (31), 11	11	4
Total				19

Table 12.13: Trace of linear probing (*N = 13*).

44 finds its home slot 5 taken by 18 and settles next door in 6. 59 gets its home slot 7. 32's home slot 6 has just been taken by 44, and 59 holds 7, so 32 ends up in 8: though its home slot is not 5, it has fallen into the crowd that starts there.

31 fares worse. Slots 5 through 9 are all occupied, so it reaches 10 after six probes. 73, home slot 8, passes 8, 9, and 10 to reach 11. The seven slots from 5 to 11 now form one solid block.

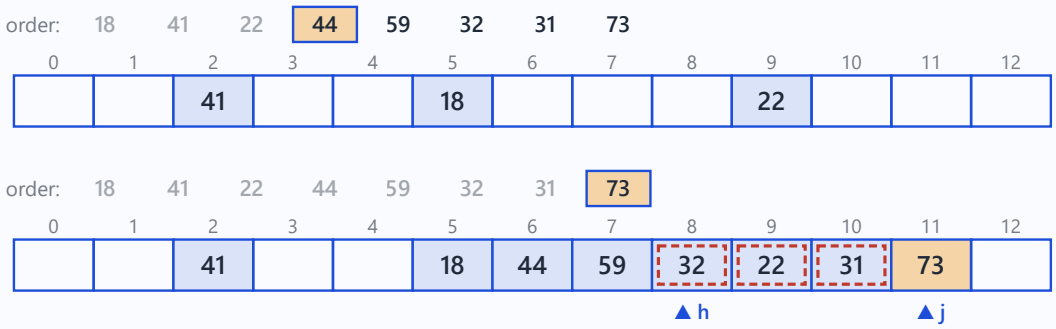


Figure 12.16: Linear probing. Top: 18 → 5, 41 → 2, 22 → 9; 44 is next. Bottom: 73's home slot is 8 (▲h); with 8, 9, and 10 occupied (dashed), it goes to 11 (▲j). Slots 5–11 form one large cluster.

Definition 12.7

Primary clustering is linear probing's tendency to form long, unbroken blocks of occupied slots that grow faster the larger they get.

Clustering feeds on itself: a key landing anywhere in a cluster walks to its end and extends it, and a longer cluster catches more keys. Neither 32 nor 73 has home slot 5, yet both joined the cluster starting there. And when the one empty slot between two clusters fills, they merge into a bigger one.

So the cost of linear probing depends on cluster length. In a sparse table almost every key goes in on the first try; as it fills, clusters lengthen and probes climb fast, as the numbers in 12.5.1 show.

12.4.3 Quadratic probing

Primary clustering comes from probing neighboring slots. Quadratic probing escapes clusters with bigger jumps, trying slots 1, 4, 9, 16, … beyond the home slot.

```
// put(k): quadratic probing, N = 13
int h = k % 13;                 // home slot
for (int i = 0; i < 13; i++) {
  int j = (h + i * i) % 13;       // h, h+1, h+4…
  if (table[j] == null) {       // empty?
    table[j] = k;
    break;
  }                             // occupied: i++
}
```

Only `j` changes: `(h + i * i) % 13` offsets the home slot by 0, 1, 4, 9 for `i = 0, 1, 2, 3`. The same keys again:

k	h	Slots tried	Final slot	Probes
18	5	5	5	1
41	2	2	2	1
22	9	9	9	1
44	5	5 (18), 6	6	2
59	7	7	7	1
32	6	6 (44), 7 (59), *6 + 4 = 10*	10	3
31	5	5 (18), 6 (44), *5 + 4 = 9* (22), *5 + 9 = 14 → 1*	1	4
73	8	8	8	1
Total				14

Table 12.14: Trace of quadratic probing ($N = 13$).

The difference starts with 32, whose third try, $6 + 2^2 = 10$, lands outside the cluster instead of extending it to 8. On its fourth try 31 jumps to $5 + 3^2 = 14$, slot $14 \bmod 13 = 1$. With slot 8 free, 73 goes home on the first try. Total probes fall from 19 to 14, the longest sequence from 6 to 4.

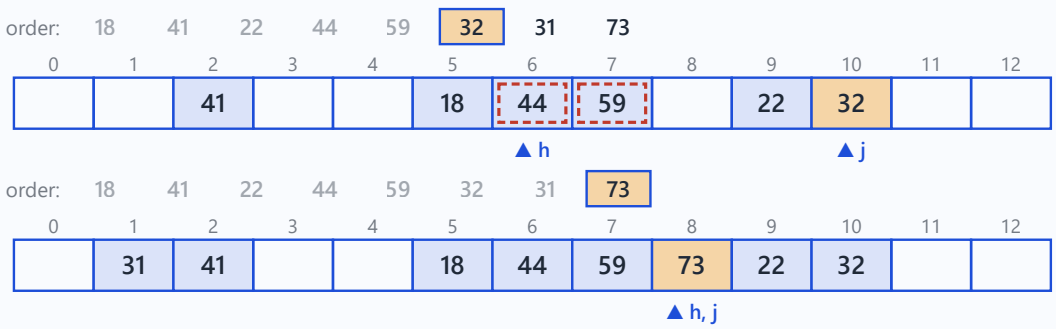


Figure 12.17: Quadratic probing. Top: inserting 32; 6 and 7 are occupied (dashed), and the third try, $6 + 4 = 10$, is empty. Bottom: all keys in; 31 jumped to slot 1, and 73 took its home slot 8 at once.

Quadratic probing reduces primary clustering but leaves another problem: keys with the same home slot follow the same sequence. 18, 44, and 31 all start at 5 and try 5, 6, 9, 1, ..., blocking one another. This is **secondary clustering**, milder than primary clustering because it affects only keys that share a home slot.

Its second weakness is that it may not try every slot. For $N = 13$, $h = 5$, the tries $i = 0, 1, \ldots, 12$ visit 5, 6, 9, 1, 8, 4, 2, 2, 4, 8, 1, 9, 6 — only seven distinct slots, {1, 2, 4, 5, 6, 8, 9}. If those are full, the key finds no room though six slots are empty. The following result avoids this.

Theorem 12.3 (An empty slot is guaranteed under quadratic probing)

If N is a prime greater than 2, the first $(N + 1)/2$ tries of quadratic probing visit distinct slots. Hence, if the table is less than half full ($\lambda < 1/2$), every insertion finds an empty slot.

Proof. Suppose tries $0 \le i < j \le (N-1)/2$ hit the same slot: $h + i^2 \equiv h + j^2 \pmod N$. Then the prime N divides $j^2 - i^2 = (j-i)(j+i)$, hence one of the factors. But $1 \le j - i < N$ and $1 \le j + i \le N - 1$, so neither is divisible by N — a contradiction. With $\lambda < 1/2$, at most $(N-1)/2$ slots are occupied, so the $(N+1)/2$ distinct slots cannot all be full. □

For $N = 13$, the first seven tries ($i = 0 \ldots 6$) are distinct — exactly the seven slots found above, after which they repeat. Hence tables using quadratic probing take a prime N and keep λ below 0.5.

12.4.4 Double hashing

Secondary clustering arises because the step depends only on the home slot. Double hashing makes it depend on the key, through a second hash function h_2; the ith try is

> $h(k, i) = (h(k) + i \cdot h_2(k)) \bmod 13,\ h_2(k) = 7 - (k \bmod 7)$

Since $k \bmod 7$ runs from 0 to 6, $h_2(k)$ runs from 1 to 7 and is never 0; a zero step would probe the home slot forever. In code:

```
// put(k): double hashing, N = 13
int h = k % 13;
int d = 7 - k % 7;              // step: 1…7
for (int i = 0; i < 13; i++) {
  int j = (h + i * d) % 13;
  if (table[j] == null) {       // empty?
    table[j] = k;
    break;
  }                             // occupied: i++
}
```

`d = 7 - k % 7` is the step; `%` binds tighter than subtraction, so it means $7 - (k \bmod 7)$. Only `j = (h + i * d) % 13` differs from the other methods.

k	$h(k)$	$d = h_2(k)$	Slots tried	Final slot	Probes
18	5	7 − 4 = 3	5	5	1
41	2	7 − 6 = 1	2	2	1
22	9	7 − 1 = 6	9	9	1
44	5	7 − 2 = 5	5 (18), *5 + 5 = 10*	10	2
59	7	7 − 3 = 4	7	7	1
32	6	7 − 4 = 3	6	6	1
31	5	7 − 3 = 4	5 (18), *5 + 4 = 9* (22), *5 + 8 = 13 → 0*	0	3
73	8	7 − 3 = 4	8	8	1

k	$h(k)$	$d = h_2(k)$	Slots tried	Final slot	Probes
Total					11

Table 12.15: Trace of double hashing ($N = 13$, $h_2(k) = 7 - (k \bmod 7)$).

Of the three keys with home slot 5, 18 goes in at once, 44 (step 5) jumps to 10, and 31 (step 4) tries 5, 9, then $5 + 8 = 13$, slot 0. Starting from the same slot, they move by different steps and stay out of each other's way. With 44 no longer in slot 6, 32 also goes home at once, and the total drops to 11.

Double hashing does not abolish secondary clustering: keys with the same home slot *and* step still share a sequence. 59, 31, and 73 all have step 4 but different home slots, so they do not interfere. Both functions colliding at once is far less likely than one.

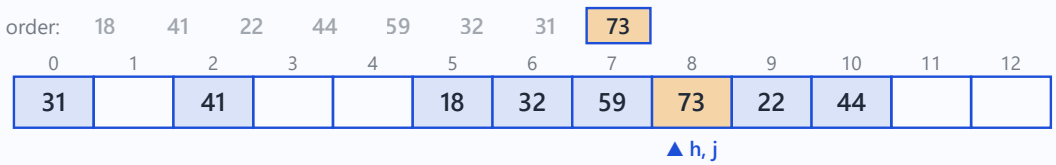


Figure 12.18: The table after double hashing: 31 → 0, 41 → 2, 18 → 5, 32 → 6, 59 → 7, 73 → 8, 22 → 9, 44 → 10. The last key, 73, took its home slot (▲h, j) on the first try.

Why pair the step with a prime N?

Theorem 12.4 (Probe sequence of double hashing)

If the step d and N are coprime, the tries $h, h + d, h + 2d, \ldots, h + (N - 1)d \pmod N$ visit each of the N slots exactly once. If N is prime, every step from 1 to $N - 1$ qualifies.

Proof. Suppose $h + i \cdot d \equiv h + j \cdot d \pmod N$ with $0 \le i < j \le N - 1$. Then N divides $(j - i) \cdot d$, and since d and N share no divisor but 1, N divides $j - i$. But $0 < j - i < N$, a contradiction. So the N tries are distinct and cover all slots. □

Here $N = 13$ is prime and the steps are 1 to 7, so any key can reach every slot. With $N = 12$ and step 4, probing would visit only $h, h + 4, h + 8$ and could miss free slots.

12.4.5 Deletion: the tombstone

Deletion under open addressing is subtler than it looks. Take the table left by linear probing — 41 in slot 2, and 18, 44, 59, 32, 22, 31, 73 in slots 5 to 11 — and delete 44.

Finding 44 is easy: $h(44) = 5$ holds 18, and slot 6 holds 44. Setting slot 6 to `null` seems natural but breaks the table. A search for 32 starts at $h(32) = 6$ and finds it empty, and under open addressing an empty slot means "the key is not further on," since had it been empty when 32 arrived, 32 would be

there. The search reports 32 absent, yet 32 is in slot 8.

The deleted slot was a link in other keys' probe sequences: 32 went to 8 because 6 and 7 were taken, and emptying 6 cuts its path. So instead of emptying the slot, we mark it.

Definition 12.8

A **tombstone** (or **DEFUNCT**) is a special value left in a deleted pair's slot under open addressing. A search treats it *as occupied* and keeps going; an insertion may reuse it *as empty*. Figures show it as ✗.



Figure 12.19: Deletion under open addressing (✗: tombstone; ○ under a cell: probed; ✓: found). (a) `remove(44)`: $h(44) = 5$, 18 ≠ 44; 44 is in slot 6. (b) Wrong: slot 6 emptied; `get(32)` stops at empty slot 6 and reports "absent," though 32 is in 8. (c) Right: a ✗ in slot 6; `get(32)` skips it, passes 59 in 7, and finds 32 in 8. (d) `put(57)`: $h = 5$; slots 5, 6 (✗), 7, ..., 11 are occupied and 12 is empty, so 57 is absent and goes into the first available slot, 6.

The last panel shows the tombstone during insertion. Inserting 57, the search notes the *X* in slot 6 as the first available slot but keeps going, since 57 might be further on. Only on reaching a truly empty slot (12) is 57 known to be absent, and only then is it written into slot 6, so the deleted slot is not wasted.

Tombstones have a cost: they pile up, and searches must pass them like occupied slots, so after many deletions probe sequences stay long even with few real items. Tables therefore count tombstones too and, when needed, rebuild from scratch, which clears them.

> **Common mistake**
>
> Setting a deleted slot to `null` under open addressing. The deletion seems to work, but later searches report present keys as absent — only keys placed beyond the deleted slot, so small tests easily miss the bug.

12.4.6 Code: findSlot()

Search and insertion probe the same way, so the shared work goes into one helper, `findSlot`, whose single integer result says whether the key was found and, if not, where it can go.

```
private int findSlot(int k) {      // probe for k
  int h = Math.floorMod(k, N), avail = -1;
  for (int i = 0; i < N; i++) {
    int j = (h + i) % N;           // linear
    if (table[j] == null ||
        table[j] == DEFUNCT) {
      if (avail == -1) avail = j;
      if (table[j] == null) break;
    } else if (table[j].key == k)
      return j;                    // found
  }
  return -(avail + 1);        // absent: encoded slot
}
```

- `h` is the home slot (`Math.floorMod` handles negative keys). `avail` holds the first empty or deleted slot seen, −1 until one appears.
- The loop probes linearly: `j = (h + i) % N`.
- An empty (`null`) or deleted (`DEFUNCT`) slot is noted in `avail` if none has been noted yet. A *truly empty* slot ends the loop, since the key cannot be further on; a deleted one does not.
- An occupied slot holding the key returns its index, 0 or more, meaning "found."
- If the loop ends without the key, `-(avail + 1)` is returned: always negative, meaning "not found," and encoding the insertion slot.

Why not `-avail`? If `avail` is 0, `-avail` is 0 too, indistinguishable from "found in slot 0." Adding 1 makes every "not found" strictly negative — slot 0 becomes −1, slot 6 becomes −7 — and the slot is recovered as *−(j + 1)*. Java's `Arrays.binarySearch` reports a missing item's insertion point the same way.

Search and insertion then take a line each:

- `get(k)`: with `j = findSlot(k)`, return `table[j].value` if `j ≥ 0`, else `null`.
- `put(k, v)`: if `j ≥ 0`, update that slot's value; otherwise write the pair into `table[-(j + 1)]`.

For quadratic probing or double hashing only `j` changes, to `(h + i*i) % N` or `(h + i*h2) % N`.

The code assumes the table is never full: $\lambda < 1$ is maintained by growing the table, so the loop always meets an empty or deleted slot and `avail` never stays −1. In a full table it would, and the result `-(-1 + 1) = 0` would be misread as "found in slot 0."

Cost. Each probe is *O(1)*, so time is proportional to the number of probes: *O(N)* in the worst case, a small constant in expectation at a low load factor.

12.4.7 Code trace: findSlot()

In the linear-probing table, 44 has been deleted, leaving ✗ in slot 6: 2 → 41, 5 → 18, 6 → ✗, 7 → 59, 8 → 32, 9 → 22, 10 → 31, 11 → 73, the rest empty. We call `findSlot(32)`, then `findSlot(57)`.

i	j	Slot	Decision	avail
—	—	—	start	−1
0	6	✗ (deleted)	first available slot; deleted ≠ empty, go on	6
1	7	59	59 ≠ 32, go on	6
2	8	32	32 = 32, `return 8`	6

Table 12.16: Trace of `findSlot(32)` (*h = 32 mod 13 = 6*).

The search does not stop at the ✗; it notes slot 6 as "insert here if needed," moves on, and returns 8, so `get` returns `table[8].value`. The tombstone kept the path intact.

i	j	Slot	Decision	avail
—	—	—	start	−1
0	5	18	18 ≠ 57, go on	−1
1	6	✗	first available slot; go on	6
2	7	59	59 ≠ 57, go on	6
3	8	32	32 ≠ 57, go on	6
4	9	22	22 ≠ 57, go on	6
5	10	31	31 ≠ 57, go on	6
6	11	73	73 ≠ 57, go on	6

`i`	`j`	Slot	Decision	`avail`
7	12	empty	`avail` stays 6; truly empty: `break`	6
Returns: *−(6 + 1) = −7*				—

Table 12.17: Trace of `findSlot(57)` (*h = 57 mod 13 = 5*).

`findSlot(57)` makes eight probes. The ✗ in slot 6 becomes `avail`; slot 12 is empty too, but `if (avail == -1)` is false, so the first available slot is kept. A truly empty slot ends the loop — 57 cannot be further on — and the method returns *−7*. `put` sees *j = −7 < 0* and sets `table[-(j + 1)] = table[6] = 57`, reusing the deleted slot.

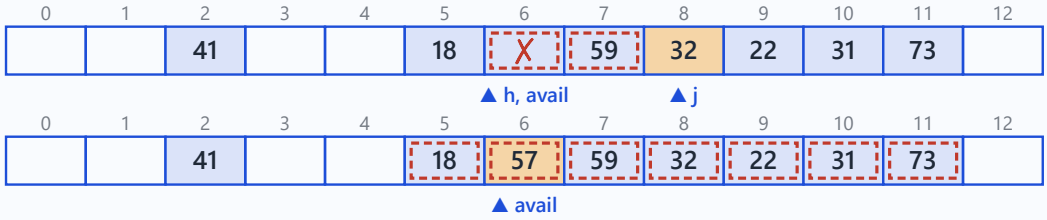


Figure 12.20: The `findSlot` trace. Top: `findSlot(32)`; ✗ in home slot 6 (▲h, avail), 32 found in 8 (▲j). Bottom: `findSlot(57)` returned −7, and `put` wrote 57 into deleted slot 6.

Writing 57 into slot 12 instead would still be correct, but the ✗ in 6 would remain and cost every search through it an extra probe. Reusing the first available slot clears tombstones over time and keeps probe sequences short.

12.4.8 The three probing methods compared

Table 12.18 sums up the three runs on the same eight keys.

Method	Total probes	Average per key	Longest sequence	Final table (slot: key)
Linear	19	2.38	6 (31)	2: 41, 5: 18, 6: 44, 7: 59, 8: 32, 9: 22, 10: 31, 11: 73
Quadratic	14	1.75	4 (31)	1: 31, 2: 41, 5: 18, 6: 44, 7: 59, 8: 73, 9: 22, 10: 32
Double hashing	11	1.38	3 (31)	0: 31, 2: 41, 5: 18, 6: 32, 7: 59, 8: 73, 9: 22, 10: 44

Table 12.18: Inserting 18, 41, 22, 44, 59, 32, 31, 73 into *N = 13* slots (*λ = 8/13 ≈ 0.62*).

This one example reflects the general trend. Linear probing clusters most; quadratic probing breaks primary clustering but keeps secondary; double hashing largely avoids both. Yet linear probing remains popular: consecutive slots are the cheapest access pattern for the cache, and at a low load factor clusters stay short, whereas double hashing needs a second hash per key and jumps far between slots.

To try the methods on other keys, use the sandbox in the slides or the open addressing simulator, which works on $N = 11$ slots with $h(k) = k \bmod 11$ and shows the slots tried, tombstones, and load factor for every insertion and search.

12.5 Performance and applications

The promise of $O(1)$ on average holds only if the table does not get too full. This section measures how fullness affects cost, then covers what happens as a table fills, how Java's `HashMap` handles it, and where hash tables are used.

12.5.1 The load factor

The load factor $\lambda = n / N$ is pairs per bucket, n being `size` in the code and N the number of buckets or slots. Its meaning depends on the strategy:

- **Under chaining**, λ is the average chain length. It may exceed 1, which only slows things down; the expected cost is $O(1 + \lambda)$.
- **Under open addressing**, $\lambda \le 1$ by necessity. As λ nears 1, empty slots grow scarce and probes multiply.

Donald Knuth derived the expected number of probes for linear probing, for a large table and a good hash function:

> successful search: $\tfrac{1}{2}\left(1 + 1/(1 - \lambda)\right)$ unsuccessful search: $\tfrac{1}{2}\left(1 + 1/(1 - \lambda)^2\right)$

An unsuccessful search runs until it reaches an empty slot, and an insertion follows the same path, so the two cost the same. The term $1 - \lambda$ is the fraction of empty slots: as it shrinks, the values grow, far faster for unsuccessful searches, where it is squared.

λ	Successful	Unsuccessful (and insertion)	Chaining $1 + \lambda$
0.25	1.17	1.39	1.25
0.5	1.5	2.5	1.5
0.75	2.5	8.5	1.75
0.8	3	13	1.8
0.9	5.5	50.5	1.9

Table 12.19: Expected probes under linear probing, against $1 + \lambda$ for chaining.

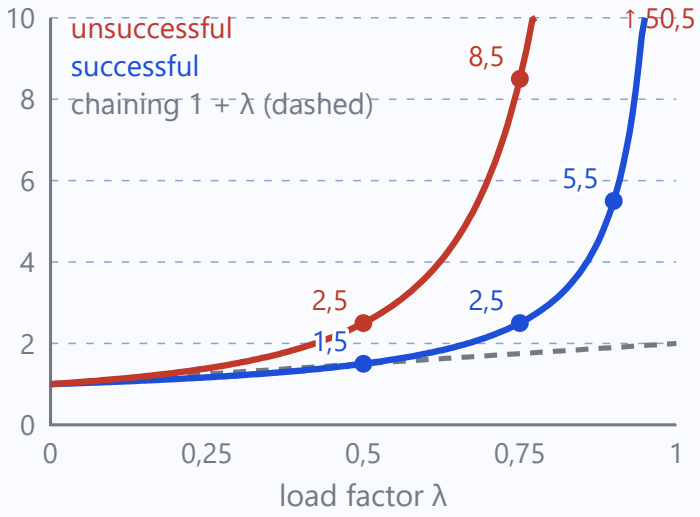


Figure 12.21: Expected probes under linear probing. Unsuccessful search (red): 2.5 at $\lambda = 0.5$, 8.5 at 0.75, 50.5 at 0.9; successful (blue): 1.5, 2.5, and 5.5. The dashed line, chaining's $1 + \lambda$, stays nearly flat.

Two lessons stand out. Under open addressing the cost is low up to $\lambda = 0.5$ and then climbs steeply: going from 50% to 80% full raises an unsuccessful search from 2.5 to 13 probes, about five times as many, and at 90% it takes 50. Under chaining the cost grows only linearly and stays acceptable even at high load.

Hence two thresholds in practice: about 0.75 for chaining (Java's `HashMap` uses it) and about 0.5 for open addressing. Past the threshold, the table roughly doubles.

Example 12.5

A linear-probing table will hold 600 keys, and an unsuccessful search should take at most 3 probes on average. How many slots are needed?

1. **Condition:** $\frac{1}{2}\left(1 + 1/(1-\lambda)^2\right) \le 3$, that is, $1/(1-\lambda)^2 \le 5$.
2. **Solution:** $(1-\lambda)^2 \ge 1/5 = 0.2$, so $1 - \lambda \ge \sqrt{0.2} \approx 0.447$ and $\lambda \le 0.553$.
3. **Slots:** $N \ge 600 / 0.553 \approx 1085.4$, so $N \ge 1086$; the first prime above 1086 is 1087, which suffices. Then $\lambda = 600 / 1087 \approx 0.552$ and an unsuccessful search takes about 3 probes.

12.5.2 Resizing

Past the threshold, the table grows. A Java array cannot change size, so growing means allocating a larger array and moving every pair into it — **resizing**. Each key is **rehashed** on the way, because its index $h(k) \bmod N$ depends on N.

In Figure 12.22, 12, 7, 20, and 33 go into a chained table of 5 buckets; 12 and 7 share bucket 2. After the fourth insertion $\lambda = 4/5 = 0.80$ exceeds 0.75, so the table grows to 11 buckets and each key moves by $k \bmod 11$: 20 → 9, 7 → 7, 12 → 1, 33 → 0. Now 7 and 12 are apart, no collisions remain, and $\lambda = 4/11 \approx 0.36$.

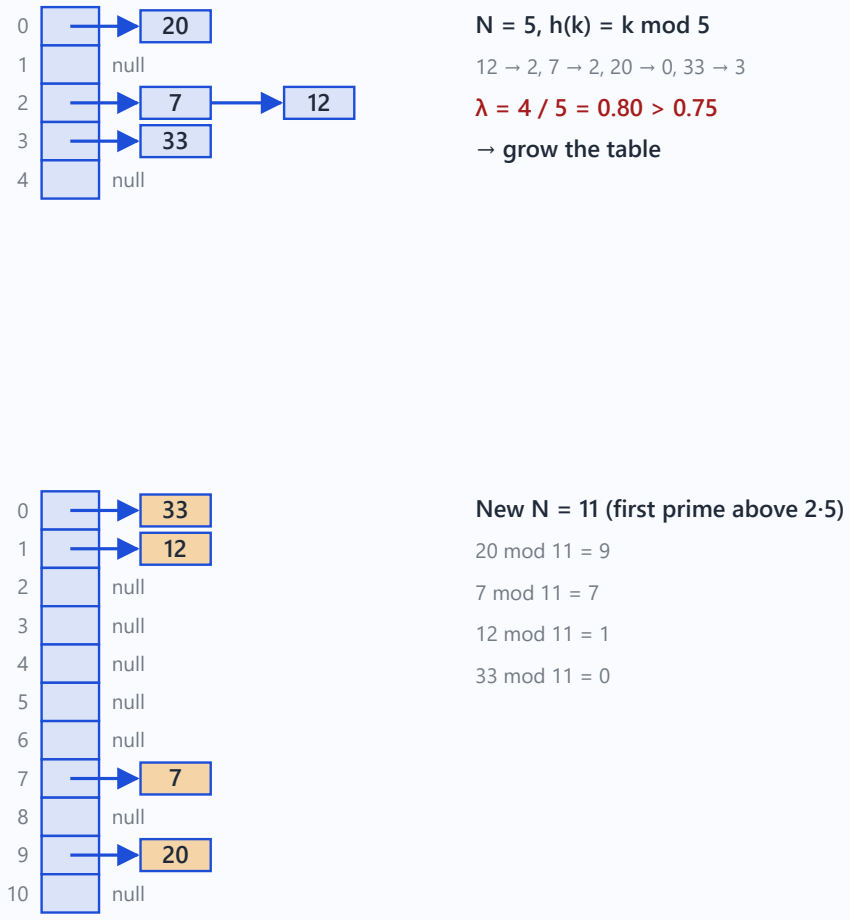


Figure 12.22: Resizing. Top: $N = 5$, $h(k) = k \bmod 5$; 12 → 2, 7 → 2, 20 → 0, 33 → 3; $\lambda = 4/5 = 0.80 > 0.75$, so the table grows. Bottom: new $N = 11$ (the first prime above $2 \cdot 5$); 20 mod 11 = 9, 7 mod 11 = 7, 12 mod 11 = 1, 33 mod 11 = 0.

Why 11 and not 10? The division method spreads keys better with a prime N (12.2.2), so some implementations pick the first prime above double the old size. The code below uses the simpler $2N + 1$, always odd: 11 for 5, a prime, though not always (7 gives 15).

A resize moves all n pairs, so by itself it costs $O(n)$. But it is rare: since capacity roughly doubles, at least as many insertions pass before the next one, and spreading its cost over them leaves a constant share each.

Theorem 12.5 (Amortized cost of resizing)

If a hash table at least doubles its capacity whenever the load factor crosses the threshold, then n insertions into an empty table move fewer than $2n$ pairs during resizes. The **amortized** cost of an insertion is therefore $O(1)$.

Proof. Let the last resize happen with $m \le n$ pairs, moving m. Because capacity at least doubles each time, the previous resize happened with at most $m/2$ pairs, the one before with at most $m/4$, and so on, for a total of $m + m/2 + m/4 + \dots < 2m \le 2n$ moves. Adding each insertion's own placement, n insertions do $O(n)$ work, a constant per insertion on average. □

Not every insertion is fast: the one that triggers a resize takes $O(n)$. Most programs care only about average speed; real-time systems, where every operation must be quick, can resize incrementally over several insertions.

12.5.3 Code trace: resize()

Here is the resize method for the chained `HashTable` class:

```
private void resize() {
  HashNode[] old = buckets;
  int newN = 2 * old.length + 1;
  buckets = new HashNode[newN];
  size = 0;  numBuckets = newN;
  for (HashNode head : old) {      // each bucket
    while (head != null) {
      put(head.key, head.value);
      head = head.next;
    }
  }
}
```

- `old = buckets` keeps the old array, the only way to reach the old chains once the new array exists.
- `newN = 2 * old.length + 1` is odd and roughly double.
- `buckets = new HashNode[newN]` points the field at the new, empty array; since `getBucketIndex` uses `buckets.length`, all later indices use the new size.
- `size = 0`, because `put` increments `size` for each pair; otherwise pairs would be counted twice.
- The outer loop visits each old bucket, the inner loop walks its chain, and `put(head.key, head.value)` places each key in its new bucket.

`put` creates fresh nodes rather than reusing old ones; once unreferenced, the old array and nodes are garbage-collected. Moving nodes directly into the new chains would be more efficient, but this version is shorter and clearer.

Tracing the previous example: the old table has bucket 0 → 20, bucket 2 → 7 → 12 (7, inserted after 12, heads the chain), and bucket 3 → 33.

Old bucket	`head`	Old index	New index	`size`
(setup)	—	—	`newN = 11`, new array empty	0
0	20	*20 mod 5 = 0*	*20 mod 11 = 9*	1
1	`null`	—	— (`while` skipped)	1
2	7	*7 mod 5 = 2*	*7 mod 11 = 7*	2
2	12	*12 mod 5 = 2*	*12 mod 11 = 1*	3
3	33	*33 mod 5 = 3*	*33 mod 11 = 0*	4
4	`null`	—	—	4

Table 12.20: Trace of `resize()` (5 buckets to 11).

Old bucket 2's chain makes the inner loop run twice — 7, then 12 after `head = head.next` — and the two keys split into new buckets 7 and 1. All four pairs end up moved, `size` is 4 again, and $\lambda = 4/11 \approx 0.36$.

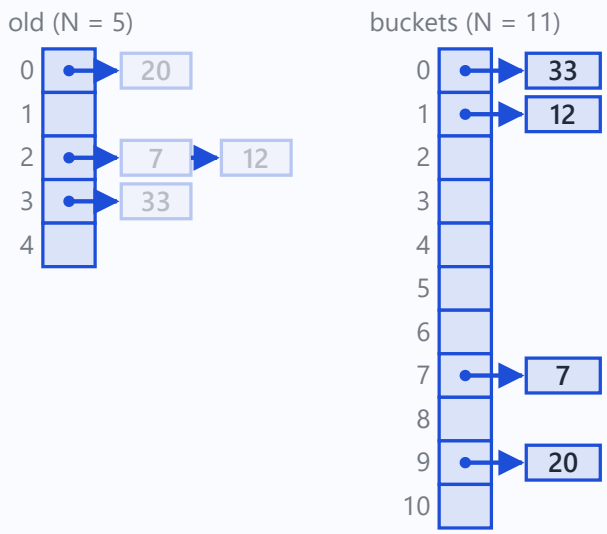


Figure 12.23: `resize()` done. Left, faded: the old table ($N = 5$); right: the new one ($N = 11$): 0 → 33, 1 → 12, 7 → 7, 9 → 20.

Cost. The outer loop runs N_{old} times and the inner loops n times in all, each `put` *O(1)* on average: *O(N + n)*, which is *O(n)* since the threshold keeps N proportional to n. Both arrays are in memory during the move.

Common mistake

Resizing by copying the old array into the new one (`System.arraycopy`, `Arrays.copyOf`). Right for a dynamic array, wrong for a hash table: 12 stays in bucket 2, but `get(12)` in the new table looks in bucket *12 mod 11 = 1* and misses it. Every key must be rehashed.

12.5.4 Complexity compared

The map ADT's implementations side by side:

Map implementation	`get`	`put`	`remove`
Unsorted array / list	O(n)	O(n)*	O(n)
Sorted array (binary search)	O(log n)	O(n)	O(n)
Hash table: average	O(1)	O(1)	O(1)
Hash table: worst	O(n)	O(n)	O(n)

Table 12.21: Cost of operations in map implementations (*n* pairs).

On the starred cell: appending to an unsorted list is *O(1)*, but map keys are unique, so we must first check whether the key exists, scanning the list in *O(n)*. In a sorted array, binary search takes *O(log n)*, but insertion and deletion shift items in *O(n)*.

Three points about hash tables stand out:

- **The worst case.** If all keys land in one bucket, one long chain or one huge cluster makes operations *O(n)*. The *O(1)* average needs a good hash function and a bounded load factor.
- **Space.** *O(n + N)* for the bucket array and the pairs, plus one reference per node under chaining. A bounded load factor gives *N = O(n)*, so space is *O(n)*.
- **No order.** Finding the smallest key, listing keys in order, or range queries such as "keys between 50 and 100" mean visiting the whole table, *O(n)* or more if sorting is needed. When order matters, search trees are the better choice.

12.5.5 Inside Java's HashMap

Java's `java.util.HashMap` combines this chapter's ideas with a few clever details.

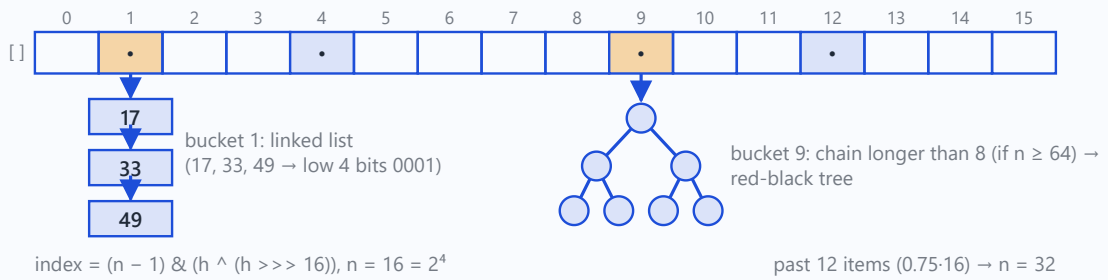


Figure 12.24: Inside Java's `HashMap`. With 16 buckets the index is `(n - 1) & (h ^ (h >>> 16))`. Bucket 1 holds a linked list: 17, 33, 49 (low 4 bits 0001). A chain longer than 8 in a table of at least 64 buckets becomes a red-black tree. Past 12 items ($0.75 \cdot 16$) the table grows to 32 buckets.

- **Chaining with a power-of-two bucket count.** The number of buckets n is always a power of 2 (16 by default), so the index comes from a fast bit operation instead of a division: `h & (n - 1)`. For $n = 16 = 2^4$, $n - 1 = 15$ is `1111` in binary, so `& 15` keeps the last four bits — the same as $h \bmod 16$ for non-negative h, and still within 0 … 15 for negative codes.
- **Bit mixing.** The mask sees only low bits, so keys differing only in high bits would always collide. `HashMap` therefore first computes `h ^ (h >>> 16)`: `>>> 16` shifts the upper 16 bits down and `^` (XOR) mixes them into the low bits, so high bits affect the index too.
- **The $\lambda = 0.75$ threshold.** When items exceed $0.75 \cdot n$, the array doubles and nodes are redistributed; with 16 buckets, the 13th item triggers growth to 32. Because n is a power of 2, each node either stays put or moves to old index + old size, decided by a single bit.
- **Treeification (Java 8+).** A chain longer than 8 in a table of at least 64 buckets becomes a red-black tree, a balanced search tree, cutting that bucket's worst-case search from $O(n)$ to $O(\log n)$. Smaller tables grow instead, since there a long chain more likely means too little capacity.

The figure's 16-bucket table is drawn to show everything at once; a real 16-bucket `HashMap` would grow rather than turn bucket 9's long chain into a tree, being below the 64-bucket limit.

Iteration order is a frequent question. Iterating with `keySet()` or a `for` loop follows bucket order, not insertion order, and resizing moves nodes, so the order can change. For insertion order use `LinkedHashMap`, for sorted keys `TreeMap`; for presence alone, `HashSet` uses the same hash table structure.

12.5.6 Code trace: index(k, 16)

To see bit mixing at work, this short method imitates `HashMap`'s index calculation, and `main` prints three keys' indices for 16 buckets.

```
static int index(Object key, int n) { // n = 2^k
  int h = key.hashCode();
  int m = h ^ (h >>> 16); // mix in the high bits
  return m & (n - 1); // low k bits
}
// main:
for (int k : new int[]{1, 65537, 131073})
  System.out.println(k + " -> " + index(k, 16));
```

For an `Integer`, `key.hashCode()` is the number itself. `h >>> 16` shifts right 16 bits, filling with zeros, so the upper half drops into the lower; `h ^ (h >>> 16)` XORs it into the low 16 bits and leaves the high 16 unchanged; `m & (n - 1)` keeps the low 4 bits.

The keys were chosen on purpose: 1, 65537, and 131073 are `0x00000001`, `0x00010001`, and `0x00020001` in hexadecimal. Their low 16 bits agree (`...0001`) and only the high bits differ, so without mixing `h & 15` would give 1 for all three.

`k` (`h`)	Hexadecimal	Unmixed `h & 15`	`h >>> 16`	`m = h ^ (h >>> 16)`	Low 4 bits	`m & 15`
1	`0x00000001`	1	0	1	0001	1
65537	`0x00010001`	1	1	65536	0000	0
131073	`0x00020001`	1	2	131075	0011	3

Table 12.22: Trace of `index(k, 16)`.

Key 1's upper half is zero, so the XOR changes nothing and the index is 1. For 65537 the upper half is 1, and `0001` XOR `0001` leaves low bits `0000`: *m = 65536*, index 0. For 131073 it is 2 (`0010`), and `0001 ^ 0010 = 0011`: *m = 131075*, index 3. The program prints:

```
1 -> 1
65537 -> 0
131073 -> 3
```

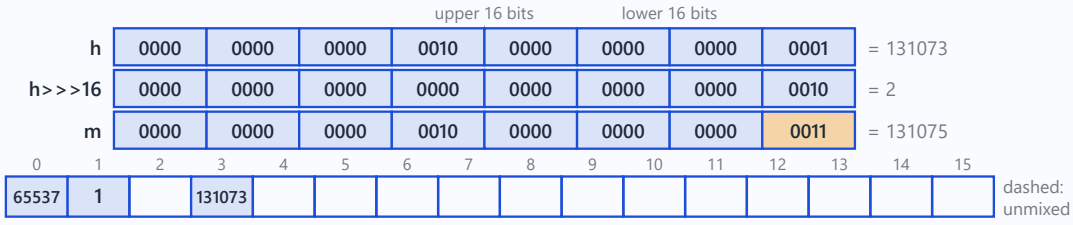


Figure 12.25: Mixing for 131073: `h`, `h>>>16`, and `m` in 4-bit groups. `m` ends in `0011`, so the key goes to bucket 3. Below, 65537 is in bucket 0, 1 in bucket 1, 131073 in bucket 3; the dashed outline marks the unmixed index (1).

Unmixed, all three keys would collide in bucket 1; mixed, they spread to 1, 0, and 3, for the price of one shift and one XOR. For keys with small hash codes ($h < 65536$) the upper half is zero and mixing changes nothing — the low bits already carry everything.

12.5.7 Where hash tables are used

Hash tables appear wherever "is this key here, and what is its value?" needs a fast answer:

- **Dictionaries and spell checkers.** A word → meaning map lives in a hash table, and a spell checker asks of every word "is it in the dictionary?", in constant time on average.
- **Databases.** Hash indexes give fast access to records by key. A *hash join*, which combines two tables on a shared column, loads one table into an in-memory hash table and looks up each row of the other.
- **In-memory caches.** The result of an expensive computation or query is stored under a key such as a URL or the query text and returned directly when the request recurs. Redis and memcached are built on this idea.
- **Browser history.** Address → time of last visit; the browser colors links by whether they have been visited.
- **Cookies and sessions.** Cookie name → value, session ID → user data; the server finds the user by session ID on every request.
- **Compilers.** The symbol table maps names to type, address, and scope, and the compiler looks up every name it meets.

In Java, most of this uses `HashMap` and `HashSet`. Small tasks — counting occurrences, finding the common items of two lists, removing duplicates — also take $O(n)$ expected time with a hash table. The next application is a typical case.

12.5.8 Application: twoSum()

Given an integer array, find the indices of two items that sum to a target. Brute force tries every pair (i, j) with nested loops, in $O(n^2)$; a hash table needs a single pass.

```
static int[] twoSum(int[] a, int target) {
  Map<Integer, Integer> seen = new HashMap<>();
  for (int i = 0; i < a.length; i++) {
    int need = target - a[i]; // the partner we need
    if (seen.containsKey(need))
      return new int[]{seen.get(need), i};
    seen.put(a[i], i); // value -> index
  }
  return null; // no such pair
}
```

At `a[i]`, the partner that completes the target is `need = target - a[i]`; if it has been seen, we are done. Seen values are kept in `seen` as value → index, and `containsKey(need)` takes *O(1)* on average. Otherwise `a[i]` joins the map, so later items can find it.

`i`	`a[i]`	`need`	`containsKey(need)`	`seen` **(after)**
0	8	4	false	{8=0}
1	3	9	false	{8=0, 3=1}
2	11	1	false	{8=0, 3=1, 11=2}
3	5	7	false	{8=0, 3=1, 11=2, 5=3}
4	7	5	**true**	— (`return [3, 4]`)

Table 12.23: Trace of `twoSum([8, 3, 11, 5, 7], 12)`.

None of the first four partners (4, 9, 1, 7) has been seen, so each value is stored with its index. At step five, `a[4] = 7` needs 5, seen at index 3, and the method returns `[seen.get(5), 4] = [3, 4]`: *a[3] + a[4] = 5 + 7 = 12*.

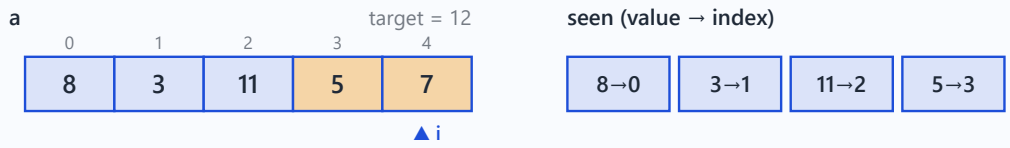


Figure 12.26: The last step of `twoSum` on [8, 3, 11, 5, 7] with target 12: at `i = 4` the partner 5 is in `seen` (5 → 3), giving [3, 4].

The check must come *before* the insertion; otherwise an item equal to half the target (6 for 12) would pair with itself. Duplicates still work: the first copy enters the map and the second finds it.

Cost. One pass, with an *O(1)*-average search and insertion per step: *O(n)* expected time and *O(n)* extra memory, against brute force's $O(n^2)$ time and no extra memory. Trading memory for time like this is the typical use of a hash table.

12.5.9 Application: hashing student IDs

General-purpose hash functions must cope with any key; knowing the keys' structure allows a much better one. Here the goal is to spread student IDs over 1000 cells with no collisions.

IDs look like `ab070c0de`, for example 210707003:

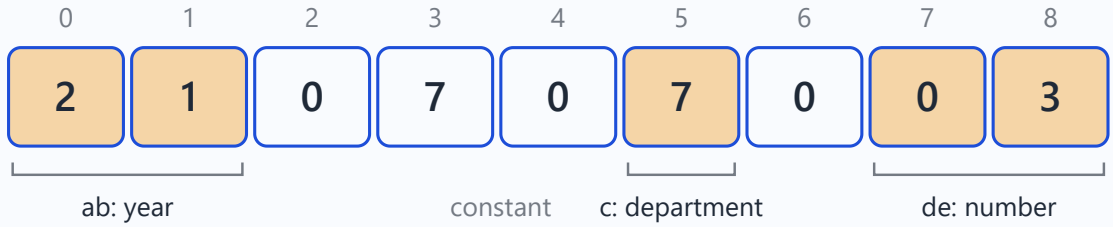


Figure 12.27: The digits of ID 210707003. 0–1: `ab`, year of enrollment (21); 2–4: constant "070"; 5: `c`, department (7); 6: constant "0"; 7–8: `de`, enrollment number (03).

- `ab`: year of enrollment, 20 to 24;
- `c`: department code, 6 or 7;
- `de`: enrollment number for that year and department, 00 to 99;
- the other digits ("070" and "0") never vary and carry no information.

This function spreads every such ID over 0–999:

> $hash(ab070c0de) = ((ab - 20) \times 100 + (c - 6) \times 500 + de) \bmod 1000$

ab − 20, from 0 to 4, picks a block of 100 (2020 → 0–99, 2021 → 100–199); *c − 6*, 0 or 1, shifts department 7 by 500; and *de*, 0 to 99, is the position in the block.

ID	`ab`	`c`	`de`	Calculation	Index
210706003	21	6	03	(1 × 100 + 0 × 500 + 3) mod 1000	103
240707050	24	7	50	(4 × 100 + 1 × 500 + 50) mod 1000	950
230707042	23	7	42	(3 × 100 + 1 × 500 + 42) mod 1000	842

Table 12.24: Indices of three student IDs.

There are *5 years × 2 departments × 100 numbers = 1000* possible IDs, each sent to its own index: department 6 fills 0–499, department 7 fills 500–999, each year its own block of 100. With no collisions at all, this is a **perfect hash**, possible only when the key set is known in advance — here, through the ID format.

In Java:

```
// s: student ID of the form "ab070c0de"
public int calculateHash(String s) {
  int ab = Integer.parseInt(s.substring(0, 2));
  int c  = Integer.parseInt(s.substring(5, 6));
  int de = Integer.parseInt(s.substring(7, 9));
  int year = (ab - 20) * 100;    // year block
  int dept = (c - 6) * 500;      // department: 0/500
  return (year + dept + de) % 1000;
}
```

`substring(begin, end)` includes `begin` and excludes `end`: `s.substring(0, 2)` is characters 0–1, the year; `s.substring(5, 6)` is character 5, the department; `s.substring(7, 9)` is characters 7–8, the enrollment number. `Integer.parseInt` converts each ("03" becomes 3), and the rest computes the formula.

Variable	"240707050"	"210706003"
`ab = substring(0, 2)`	"24" → 24	"21" → 21
`c = substring(5, 6)`	"7" → 7	"6" → 6
`de = substring(7, 9)`	"50" → 50	"03" → 3
`year = (ab - 20) * 100`	400	100
`dept = (c - 6) * 500`	500	0
Returns `(year + dept + de) % 1000`	950	103

Table 12.25: Trace of `calculateHash`.

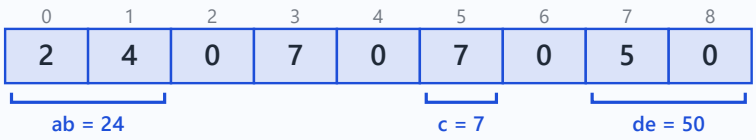


Figure 12.28: `calculateHash("240707050")`: `ab = 24` (characters 0–1), `c = 7` (character 5), `de = 50` (characters 7–8); result *400 + 500 + 50 = 950*.

The method does a fixed amount of work, *O(1)*, but is correct only for IDs in the expected format. 250707001, with year 25, gives *(500 + 500 + 1) mod 1000 = 1* and collides with 200706001; a department code of 8 overshoots 1000 and wraps onto another ID's index; a year below 20 can even give a negative result. A real program validates input first, rejecting malformed IDs or sending them to a general-purpose hash table.

Common mistake

Treating `substring`'s end index as inclusive. `s.substring(5, 6)` returns one character; `s.substring(5, 5)` returns the empty string, and `Integer.parseInt("")` throws a `NumberFormatException`. For one character, the end is the start plus one.

12.6 Common mistakes

The same mistakes recur in programs that use hash tables, and most pass the compiler and surface only on particular inputs. Table 12.26 collects those met in this chapter.

Mistake	Consequence	Correct approach
`Math.abs(h) % N`	`Math.abs(Integer.MIN_VALUE)` is negative; index out of bounds.	`Math.floorMod(h, N)`
Overriding only `equals`	Equal keys land in different buckets; `get` returns `null`.	Override `equals` and `hashCode` on the same fields, or use a `record`.
Mutable key	A key whose hashed fields change stays in its old bucket and is lost.	Immutable keys: `String`, `Integer`, `record`.
Comparing `Integer`s with `==`	Boxed values outside −128 … 127 are separate objects; `==` gives `false`.	`equals`
Deleting with `null` under open addressing	The probe path breaks; keys beyond it seem absent.	A tombstone (DEFUNCT)
Expecting an order	`HashMap` iterates in no insertion order, and resizing can change it.	`LinkedHashMap` (insertion order) or `TreeMap` (key order)
Copying the array when resizing	Keys stay at old indices and are missed under the new size.	Rehash every key.
Reading `get`'s `null` as "key absent"	`HashMap` can store `null` values.	`containsKey(k)`

Table 12.26: Common mistakes with hash tables.

Mutable keys are especially insidious. Use an `ArrayList` as a key, then add to the list, and its `hashCode()` changes: [1, 2] hashes to 994, [1, 2, 3] to 30817. `get` then searches another bucket and returns `null`, though the pair is still there and `size()` is 1. The pair can be neither found nor removed — unreachable garbage inside the table.

Most of these share a root. A hash table assumes that a key's hash code does not change while it is stored and that equal keys hash alike. Break either and the table raises no error; it just answers wrongly. So in code using a hash table, check the key class's `equals` and `hashCode` first.

12.7 Summary

- A **map** associates unique keys with values through `put`, `get`, and `remove`; `put` on an existing key updates it and returns the old value. A map is an ADT, most often implemented by a hash table.
- **Direct addressing** gives every possible key a cell and runs in *O(1)*, but wastes memory on large key spaces and cannot take non-integer keys.
- A **hash function** turns a key into a **hash code** and, by **compression**, into an index in 0 … *N* − *1*. A good one is deterministic, fast, and uniform, and uses the whole key. The division method wants a prime *N*; MAD is *((a·k + b) mod p) mod N*. In Java, index with `Math.floorMod(h, N)`.
- **The `equals` / `hashCode` contract:** equal objects must hash alike; equal hashes do not imply equality. `hashCode` picks the bucket, `equals` decides equality, and both are overridden together.
- **Collisions** are unavoidable whenever the key space exceeds the table (the pigeonhole principle).
- **Separate chaining** keeps a linked list per bucket, with new nodes possibly at the front. Operations take *O(1 + chain length)*, *O(1 + λ)* expected, and $\lambda > 1$ is allowed. Deletion tracks the previous node.
- **Open addressing** stores pairs in the array, so $\lambda \le 1$. **Linear probing** suffers primary clustering, **quadratic probing** secondary clustering; the latter guarantees an empty slot with a prime *N* and $\lambda < 1/2$. **Double hashing** ties the step to the key, which must be nonzero and coprime with *N*. Deletion leaves a **tombstone**.
- The **load factor** is $\lambda = n/N$. An unsuccessful linear-probing search averages $\frac{1}{2}(1 + 1/(1 - \lambda)^2)$ probes: 2.5 at $\lambda = 0.5$, 13 at 0.8. Past the threshold (≈ 0.75 for chaining, ≈ 0.5 for open addressing) the table roughly doubles and every key is rehashed — *O(n)* once, amortized *O(1)* per insertion.
- Insertion, search, and deletion are ***O(1)* on average, *O(n)* in the worst case**; space is *O(n + N)*; there is no order.
- **Java's `HashMap`**: chaining, a power-of-two bucket count (16 by default), mixing with `h ^ (h >>> 16)` and the mask `& (n - 1)`, doubling at 0.75, and red-black trees for chains longer than 8 (given at least 64 buckets).
- Known key structure allows a **perfect hash**: in the student ID example, 1000 possible IDs map to 1000 different indices.

Method	Where pairs live	λ limit	Weakness	Deletion
Separate chaining	Linked lists in buckets	can be > 1 (≈ 0.75 in practice)	Extra reference per node; scattered memory	Node unlinked
Linear probing	The array	≤ 1 (≈ 0.5 in practice)	Primary clustering	Tombstone
Quadratic probing	The array	< 0.5 (prime *N*)	Secondary clustering; may miss slots	Tombstone
Double hashing	The array	≤ 1 (≈ 0.5 in practice)	Second hash; not cache-friendly	Tombstone

Table 12.27: Collision resolution methods compared.

12.8 Key terms

Each term links to the course glossary, which gives its Turkish equivalent and its use in other chapters.

map · key · value · direct addressing · hash table · hash function · hash code · compression function · bucket · collision · separate chaining · open addressing · probing · linear probing · quadratic probing · double hashing · primary clustering · load factor · resizing (rehashing) · amortized analysis

12.9 Test Yourself

1. 8, 13, 3, 21, and 6 are inserted in that order into a chained table of 5 buckets with $h(k) = k \bmod 5$, new nodes at the front. Give the final table and λ. How many key comparisons does `get(8)` make?

   Answer

   Buckets: 8 → 3, 13 → 3, 3 → 3, 21 → 1, 6 → 1. The latest key comes first: bucket 1 is 6 → 21, bucket 3 is 3 → 13 → 8, and buckets 0, 2, 4 are empty. $\lambda = 5/5 = 1$. `get(8)` examines 3, 13, and 8: 3 comparisons.

2. Compute "Can".hashCode() with the `String.hashCode()` formula (C = 67, a = 97, n = 110). Which of 13 buckets does it land in?

   Answer

   $h = 67$; $31 \cdot 67 + 97 = 2174$; $31 \cdot 2174 + 110 = 67504$. $13 \cdot 5192 = 67496$, remainder 8: bucket 8, where it collides with "Ali".

3. 4, 15, 26, and 5 are inserted in that order into an empty linear-probing table of 11 slots with $h(k) = k \bmod 11$. Where does each go, and with how many probes?

   Answer

   4 → 4 (1 probe). 15: home 4 taken, 5 free (2 probes). 26: home 4; 4, 5 taken, 6 free (3 probes). 5: home 5; 5, 6 taken, 7 free (3 probes). 9 probes in all; slots 4–7 form one cluster.

4. In that table, 15 is deleted by setting slot 5 to `null`. What does `get(26)` find? What if a

tombstone is used instead?

Answer

With `null`: 4 (4 ≠ 26), then empty slot 5, so "26 absent," though 26 is in slot 6. With a tombstone: 4, 5 (✗, skipped), and 26 is found in 6 after 3 probes.

5. `findSlot` returned −1. What does it mean? With `-avail` instead of `-(avail + 1)`, which cases would be confused?

Answer

Negative means "key absent," with insertion slot $-(-1 + 1) = 0$. With `-avail`, an available slot 0 would return 0, the same as "found in slot 0."

6. An 11-slot table uses double hashing with $h(k) = k \bmod 11$ and $h_2(k) = 7 - (k \bmod 7)$. Where do 13, 24, and 35 go in an empty table? And under linear probing?

Answer

All three have home slot 2. 13 → 2. 24: step $7 - 3 = 4$, so $2 + 4 = 6$. 35: step $7 - 0 = 7$, so $2 + 7 = 9$. Linear probing would put them in 2, 3, 4, forming a cluster.

7. Keys are inserted one by one into a default Java `HashMap`. On which insertion does it first grow, and to how many buckets? When does it grow next?

Answer

16 buckets, threshold $0.75 \cdot 16 = 12$: the 13th insertion grows it to 32. The new threshold is $0.75 \cdot 32 = 24$, so the 25th insertion grows it to 64.

8. Compute $((ab - 20) \times 100 + (c - 6) \times 500 + de) \bmod 1000$ for IDs 220706099 and 220707099.

Answer

Both have $ab = 22$, $de = 99$. 220706099: $c = 6$, $200 + 0 + 99 = 299$. 220707099: $c = 7$, $200 + 500 + 99 = 799$.

9. A linear-probing table has $\lambda = 2/3$. By Knuth's formulas, how many probes do successful and unsuccessful searches average?

Answer

$1 - \lambda = 1/3$. Successful: $\frac{1}{2}(1 + 3) = 2$. Unsuccessful: $\frac{1}{2}(1 + 9) = 5$.

10. A `Point` class's `equals` compares both coordinates, and its `hashCode` returns `x + y`. Is the contract met? After `put`s with keys `new Point(1, 2)` and `new Point(2, 1)` into an empty `HashMap`, what is `size()`?

    Answer

    Yes: equal points have equal coordinates, hence equal sums. (1, 2) and (2, 1) share hash code 3 but are unequal, so they share a bucket as separate pairs: `size()` is 2. The function is correct but weak, since all points with the same sum collide.

12.10 Further reading

- M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, *Data Structures and Algorithms in Java* (6th ed., 2014), Chapter 10: the map ADT, hash codes versus compression, the MAD method, and Java implementations with chaining and linear probing, organized much like these notes.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms* (4th ed., 2022), Chapter 11: probabilistic proofs of the expected costs of chaining and open addressing, and universal hashing.
- R. Sedgewick and K. Wayne, *Algorithms* (4th ed., 2011), Section 3.4: symbol tables with chaining and linear probing in Java, with experiments on the effect of the load factor.
- D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching* (2nd ed., 1998), Section 6.4: the source of the linear-probing formulas, with a history of hashing.

12.11 References

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022.
2. M. T. Goodrich, R. Tamassia, M. H. Goldwasser, *Data Structures and Algorithms in Java*, 6th ed., Wiley, 2014.
3. D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd ed., Addison-Wesley, 1998.
4. Oracle, *Java SE 17 API Specification*: documentation for `java.util.Map`, `java.util.HashMap`, and `java.util.HashSet`.
5. R. Sedgewick, K. Wayne, *Algorithms*, 4th ed., Addison-Wesley, 2011.

PART IV · CHAPTER 13

Sets and Disjoint Sets

Is a name written twice on a class list? Is this user banned? Are two cities on the same road network? Each is a version of "is this item in that collection?", and sets are built to answer it fast. These notes start with the set and Java's `Set` interface, then compare four implementations — hash table, balanced search tree, sorted array, bit vector — and their costs. Multisets, which count repeats, and multimaps, which attach several values to a key, follow. The second half covers disjoint sets: the Union-Find structure, which splits items into non-overlapping groups and merges groups in nearly constant time, with union by size and by rank, path compression, and counting connected components.

Code for this chapter: `github.com/sercankulcu/data-structures-java/Bolum13/src`

LEARNING OUTCOMES

The set model
Explain the set ADT and Java's `Set` interface; compute union, intersection, difference, and subset both by hand and with `Set` methods.

Implementations
Trace hash-based, tree-based, sorted-array, and bit-vector implementations, and choose the right one for a problem by its cost.

Multisets
Use multisets and multimaps for problems that need repeated items.

Union-Find
Represent disjoint sets with a parent array; apply path compression and union by rank (or size), and explain their effect.

Applications
Use Union-Find for connectivity problems such as counting connected components.

13.1 The set ADT

A data structure can be described at two levels: what it does — its operations and their effects — and how, meaning its memory layout and the steps of each operation. A description of the first kind is an **abstract data type** (ADT). This section treats the set as an ADT; hash tables, trees, and bit vectors come in Section 13.2.

13.1.1 What is a set?

Think of a class roll. Each name appears once — writing it twice adds nothing — and the order doesn't matter. The only question is "Is Alice in this class?" The mathematical set captures exactly this idea.

Definition 13.1

A **set** is a collection of distinct elements. No element appears twice, and order is irrelevant: *{1, 2, 3} = {3, 1, 2}*. The basic operations of the set ADT are adding an element, removing an element, and the **membership test**: "is x in the set?"

Two rules stand out. First, no *duplicates*: adding an element already present leaves the set unchanged, whereas a list accepts the same value any number of times. Second, sets are *unordered*: there is no "first" or "third" element, so the ADT has no indexed access. Some implementations visit elements in a particular order, but that is a by-product of the implementation, not part of the definition.

Above all, a set must answer membership queries quickly. An array answers by scanning, in *O(n)*; a good set implementation takes constant time on average, or *O(log n)*. Most of this chapter is about how.

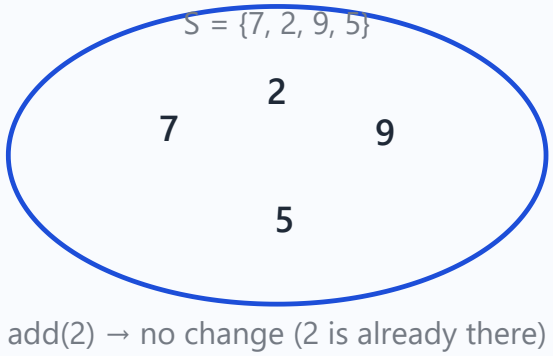


Figure 13.1: The set *S* of the elements 7, 2, 9, and 5, drawn in no particular order. Trying to add 2, which is already present (`add(2)`), leaves the set unchanged.

Figure 13.1 shows both rules: `add(2)` on *S = {7, 2, 9, 5}* finds 2 already there, so the size stays 4, and the elements are scattered because a set has no "before" or "after."

Sets do three kinds of everyday work:

- **Removing duplicates.** Add every word of a text to a set and only the distinct words remain. In Java a list becomes a set in one line: `new HashSet<>(list)`.
- **Membership.** Banned users, visited nodes, and the valid words of a dictionary are sets, queried on every request.
- **Set algebra.** Mutual friends are the intersection of two friend sets; a site's tags are the union of its posts' tag sets.

> **Common mistake**
>
> Treating a set as a sorted list. A printed `HashSet` of small integers often comes out in increasing order, but only by coincidence: elements go into buckets by hash code, and a small integer's hash code is itself. With strings or large numbers the order breaks. If you need order, use a sorted set (`TreeSet`).

13.1.2 Set operations

Four basic operations, or set **operators**, are defined on sets *A* and *B*:

- **Union**, $A \cup B$: the elements in A or in B (or in both).
- **Intersection**, $A \cap B$: the elements in both A and B.
- **Difference**, $A - B$: the elements in A but not in B.
- **Subset**, $A \subseteq B$: every element of A is also in B. This is a yes-or-no question rather than an operation.

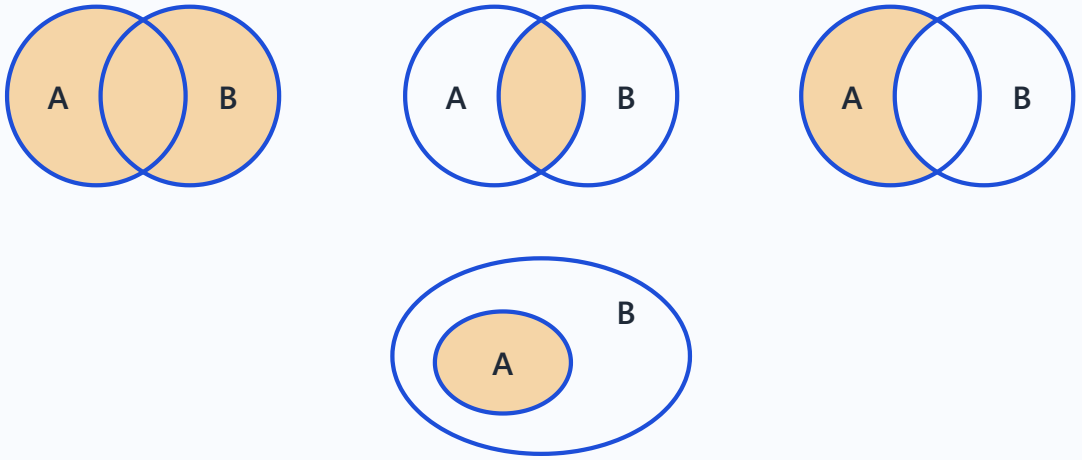


Figure 13.2: Venn diagrams of the four set operations (left to right). (a) Union $A \cup B$: both circles shaded. (b) Intersection $A \cap B$: only the common region shaded. (c) Difference $A - B$: the part of A not shared with B shaded. (d) Subset $A \subseteq B$: circle A lies inside B.

Figure 13.2 draws them as Venn diagrams, each set a circle containing its elements: a union shades both circles, an intersection only the lens-shaped overlap, a difference cuts the overlap out of A, and a subset lies entirely inside the other circle.

Union and intersection are commutative — $A \cup B = B \cup A$, $A \cap B = B \cap A$ — but difference is not: $A - B$ and $B - A$ usually differ.

Example 13.1

For $A = \{1, 2, 3, 4, 5\}$ and $B = \{3, 4, 5, 6, 7\}$, find the union, the intersection, both differences, and answer the subset questions.

1. **Union.** All of A, plus the elements of B not in A (6 and 7); 3, 4, 5 are written once. $A \cup B = \{1, 2, 3, 4, 5, 6, 7\}$.
2. **Intersection.** Of A's elements, 1 and 2 are not in B; 3, 4, 5 are. $A \cap B = \{3, 4, 5\}$.
3. **Difference.** Remove from A what is in B: $A - B = \{1, 2\}$. The other way round, $B - A = \{6, 7\}$.
4. **Subset.** Is $A \subseteq B$? No: 1 is in A but not in B ($1 \notin B$), and one counterexample suffices, so $A \not\subset B$. But $\{3, 4\} \subseteq A$, since both 3 and 4 are in A.
5. **Check.** $|A| + |B| - |A \cap B| = 5 + 5 - 3 = 7$, the size of the union.

Operation	Result
Union $A \cup B$	{1, 2, 3, 4, 5, 6, 7}
Intersection $A \cap B$	{3, 4, 5}
Difference $A - B$ / $B - A$	{1, 2} / {6, 7}

Operation	Result
Subset?	$A \not\subset B$ ($1 \notin B$) · $\{3, 4\} \subseteq A$ ✓

Table 13.1: Results of the set operations for $A = \{1, 2, 3, 4, 5\}$, $B = \{3, 4, 5, 6, 7\}$.

The check uses the *inclusion–exclusion principle*, $|A \cup B| = |A| + |B| - |A \cap B|$: common elements are counted in both sizes, so they are subtracted once.

13.1.3 The Set interface in Java

In Java the set ADT is the interface `java.util.Set<E>`, implemented by `HashSet`, `LinkedHashSet`, and `TreeSet`. `Set` extends `Collection` with one contract: no two elements are equal, as decided by `equals` (in a sorted set, by `compare`).

Method	Description
`add(e)`	adds e; if e is already present, the set is unchanged and the method returns `false`.
`remove(o)`, `contains(o)`	removes the element; tests membership.
`addAll(T)`	union: $S \leftarrow S \cup T$
`retainAll(T)`	intersection: $S \leftarrow S \cap T$
`removeAll(T)`	difference: $S \leftarrow S - T$
`containsAll(T)`	subset question: is $T \subseteq S$?

Table 13.2: The main methods of `java.util.Set` and the set operations they correspond to.

A small but valuable detail: `add` reports whether it added anything, returning `true` for a new element and `false` for one already there. That single bit is the basis of the duplicate finder in Section 13.1.4. Likewise, `remove` returns `true` only if it removed something.

The set operations are `addAll`, `retainAll`, and `removeAll`. This code repeats Example 13.1:

```
Set<Integer> a = new HashSet<>(List.of(1,2,3,4,5));
Set<Integer> b = new HashSet<>(List.of(3,4,5,6,7));
Set<Integer> uni = new HashSet<>(a);
uni.addAll(b);     // [1..7]
Set<Integer> both = new HashSet<>(a);
both.retainAll(b); // [3, 4, 5]
Set<Integer> diff = new HashSet<>(a);
diff.removeAll(b); // [1, 2]
a.containsAll(b);  // false
```

The first two lines build two `HashSet`s. The third copies `a` into `uni`, and `uni.addAll(b)` adds B's elements: 3, 4, 5 are already there, 6 and 7 are new, giving 1 through 7. For the intersection, `retainAll(b)` on another copy deletes everything not in B, leaving `[3, 4, 5]`; for the difference,

`removeAll(b)` deletes everything in B, leaving `[1, 2]`. Finally, `a.containsAll(b)` asks whether $B \subseteq A$; 6 is not in A, so the answer is `false`.

Their cost depends on the implementation. In a `HashSet`, `uni.addAll(b)` does one `add`, *O(1)* on average, per element of B, plus *O(|A|)* for the copy; `retainAll` and `removeAll` call `contains` on the other set for each element of the copy. So with hash sets, union, intersection, and difference take *O(n + m)* on average.

Common mistake

Expecting `addAll`, `retainAll`, or `removeAll` to return a new set. They **modify their set in place** and return only a `boolean` saying whether it changed. `Set<Integer> both = a; both.retainAll(b);` makes no copy: `both` and `a` are the same object, so `a` becomes the intersection. Copy first: `new HashSet<>(a)`.

13.1.4 Code trace: the first repeat via add

Suppose we want the first value in an array, read left to right, that has appeared before. Without a set, each element is compared with all earlier ones: two nested loops and up to $n(n - 1)/2$ comparisons, $O(n^2)$. With a set, we collect the elements seen so far and test each new one with a single `add`:

```
int firstRepeat(int[] a) {
  Set<Integer> seen = new HashSet<>();
  for (int x : a) {
    if (!seen.add(x))   // not added?
      return x;            // already there: a repeat
  }
  return -1;               // no repeats
}
```

The loop body does two jobs in one line. `seen.add(x)` adds a new element and returns `true`; the `!` makes the condition false, and the loop continues. For an element seen before, `add` returns `false` without changing the set, the condition holds, and the method returns that element. With no repeats, the loop ends and the method returns `-1`.

This is shorter than `contains` followed by `add`, and it searches the bucket once instead of twice.

Let's trace `a = [4, 7, 2, 7, 4]`. `HashSet` uses 16 buckets, and an element's bucket is the last four bits of its hash code, `hash(x) & 15`. An `Integer`'s hash code is its value, so 4 goes to bucket 4, 7 to bucket 7, and 2 to bucket 2.

Pass	x	Bucket	add(x)	seen (after)	Result
1	4	4	true	{4}	loop continues
2	7	7	true	{4, 7}	loop continues

Pass	`x`	Bucket	`add(x)`	`seen` (after)	Result
3	2	2	`true`	{2, 4, 7}	loop continues
4	7	7	`false`	{2, 4, 7}	**returns 7**

Table 13.3: Trace of `firstRepeat([4, 7, 2, 7, 4])`. The bucket is `x & 15`.

In the first three passes the buckets are empty, so each `add` succeeds after one bucket check. In the fourth, bucket 7 already holds 7: `add` returns `false` and the method returns 7. The last element, 4, is never examined.

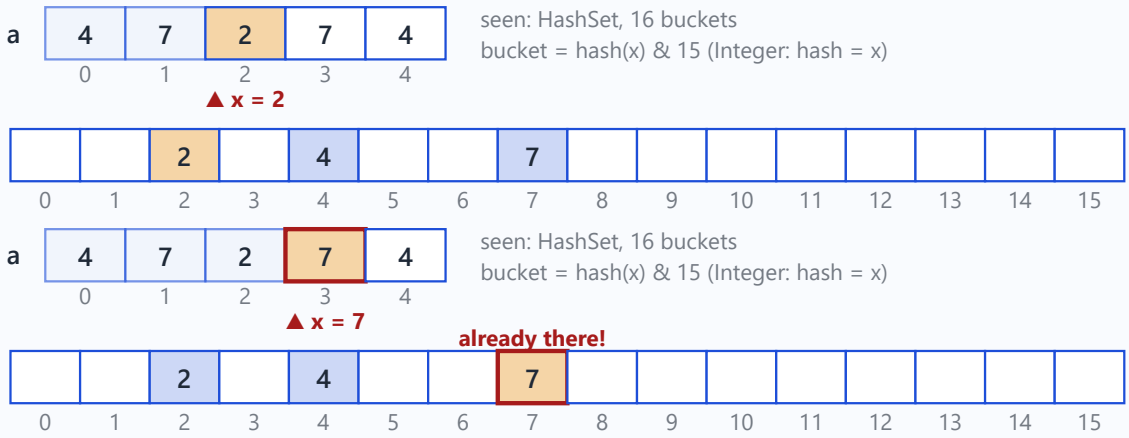


Figure 13.3: Two moments in the trace of `firstRepeat`. Top, the third pass: 2 has been added to bucket 2, and the set is {2, 4, 7}. Bottom, the fourth pass: for `x = 7`, bucket 7 already holds 7, so `add` returns `false` and the method returns 7.

Each element costs one `add`, *O(1)* on average, so the method runs in *O(n)* on average with no nested loop. The price is up to *O(n)* extra memory for the set — the typical trade of memory for time.

13.2 Implementing sets

The set ADT says only "what"; each implementation makes some operations fast and others slow. This section examines four — hash table, balanced binary search tree, sorted array, and bit vector — and ends with a table on which to choose when.

13.2.1 Set classes in Java

Java provides three `Set` classes. All keep the same contract — no duplicates; `add`, `remove`, `contains` — and differ in internal structure and hence in iteration order.

Class	Internal structure	Iteration order	`add` / `contains` / `remove`
`HashSet`	hash table	depends on hash codes, unpredictable	O(1) on average

Class	Internal structure	Iteration order	`add` / `contains` / `remove`
`LinkedHashSet`	hash table + doubly linked list	insertion order	O(1) on average
`TreeSet`	red-black tree	sorted (numeric, alphabetical)	O(log n)

Table 13.4: Implementations of `java.util.Set`.

To see the difference, add the same four words to all three:

```
for (String s : List.of("lemon", "peach", "fig", "lemon")) {
    hash.add(s);  linked.add(s);  tree.add(s);
}
// HashSet       → [lemon, fig, peach]   (bucket order: depends on hash)
// LinkedHashSet → [lemon, peach, fig]   (insertion order)
// TreeSet       → [fig, lemon, peach]   (alphabetical)
```

"lemon" is added twice but appears once: the second `add("lemon")` returns `false` in all three sets. The orders differ, though. `HashSet` lists elements bucket by bucket; in a 16-bucket table "lemon" lands in bucket 2, "fig" in 5, and "peach" in 8, giving `[lemon, fig, peach]` — neither insertion nor alphabetical order, and liable to change when the table is resized. `LinkedHashSet` also threads its elements on a doubly linked list and iterates over it, giving insertion order. `TreeSet` keeps a search tree and returns elements sorted, by an inorder traversal.

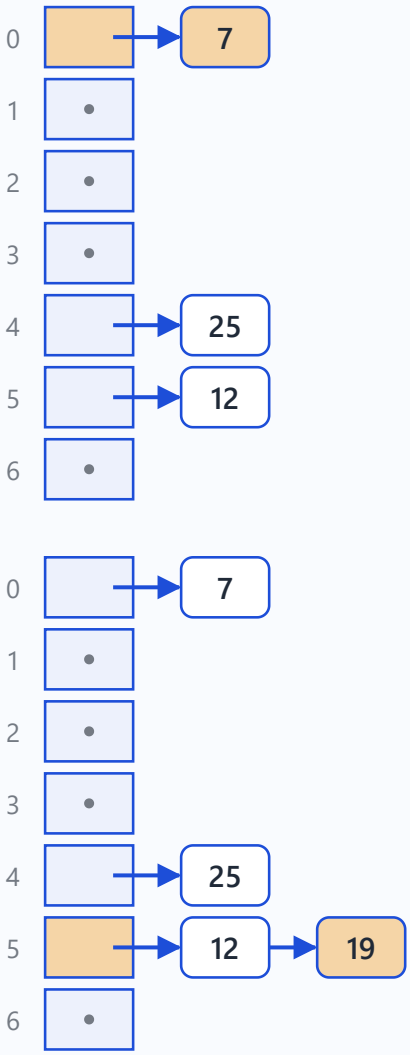
Hence the rule: `HashSet` when order doesn't matter, `LinkedHashSet` for insertion order, `TreeSet` for sorted access. `LinkedHashSet` pays for order with two extra references per element; `TreeSet` pays in time, *O(log n)* instead of *O(1)*.

13.2.2 Hash-based sets

A hash-based set stores its elements in a **hash table**, an array of m cells called **buckets**. An element's bucket is computed from its **hash code**, an integer derived from the element (equal elements have equal hash codes), usually as $i = hashCode(x) \bmod m$.

A map's hash table keeps a key–value pair per entry; a set needs only the keys. Java's `HashSet` in fact wraps a `HashMap`, storing each element as a key with the same dummy object as its value.

Two elements landing in the same bucket is a **collision**: with 7 buckets, 12 and 19 both go to bucket 5, since $12 \bmod 7 = 19 \bmod 7 = 5$. The simplest remedy is **separate chaining**: each bucket holds a list (a chain) of the elements that land there.


0
7
1
2
3
4
25
5
12
6
0
7
1
2
3
4
25
5
12
19
6

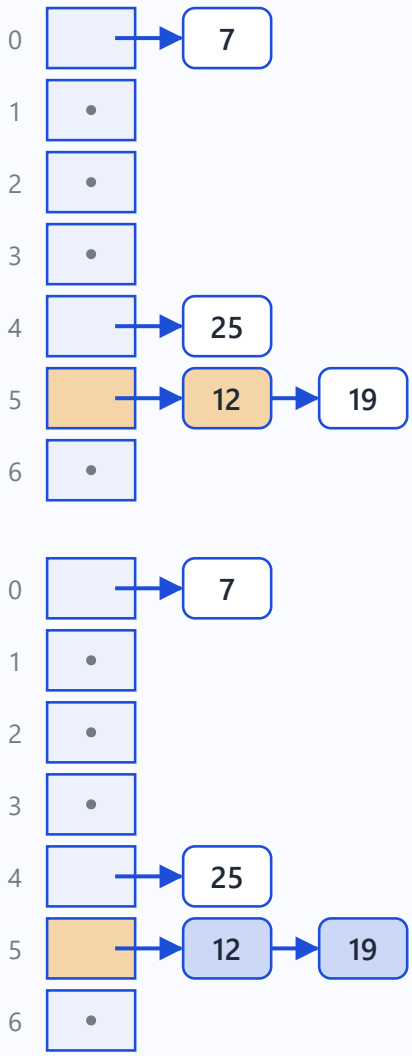


Figure 13.4: A chained hash set with 7 buckets; position *hashCode(x) mod 7*. (a) After adding 12, 25, and 7: 12 in bucket 5, 25 in bucket 4, 7 in bucket 0. (b) `add(19)` : 19 also lands in bucket 5 and goes to the end of the chain. (c) `add(12)` again: bucket 5 is scanned, 12 is found, and nothing is added. (d) `contains(26)` : only bucket 5 (12, 19) is scanned; 26 is not found.

Figure 13.4 adds 12, 25, 7, 19, and 12 again to an empty 7-bucket table. 12 goes to bucket 5, 25 to bucket 4 (*25 mod 7 = 4*), and 7 to bucket 0. 19 also maps to bucket 5, which holds 12 — a collision — so 19 is appended to the chain. The second `add(12)` scans bucket 5, finds 12, and returns `false` without adding anything. This check is what makes the structure a set.

Hashing is fast because `contains(26)` looks only at bucket *26 mod 7 = 5*: it compares 26 with 12 and 19 and returns `false` , never touching buckets 0 and 4. However large the table, a search is confined to one chain.

How long are the chains? If n elements spread evenly over m buckets, the average length is n / m, the **load factor** λ; here $\lambda = 4/7 \approx 0.57$. When λ exceeds a threshold (0.75 in Java), the table roughly doubles its buckets and redistributes the elements, so λ stays bounded and `add` , `contains` , and `remove` take *O(1)* on average.

This assumes a hash function that spreads elements well. One that sends everything to the same bucket turns the table into a single chain, and every operation costs *O(n)*.

> **Common mistake**
>
> Overriding `equals` without `hashCode`. Two "equal" objects then almost certainly get different hash codes and land in different buckets. `HashSet` compares with `equals` only within a bucket, so it stores both, and the set holds the same element **twice**. Override the two together, so equal objects get equal hash codes.

13.2.3 Code trace: add and contains

The two basic methods of a chained hash set are short. The fields are `m`, the number of buckets; `bucket`, an array of integer lists; and `n`, the number of elements. Assuming non-negative elements, the bucket number is simply `x % m`.

```
// fields: List<Integer>[] bucket; n; m = 7
boolean add(int x) {          // x ≥ 0
  int i = x % m;               // bucket no.
  for (int y : bucket[i])
    if (y == x) return false;  // already there
  bucket[i].add(x);
  n++;
  return true;
}
boolean contains(int x) {
  int i = x % m;
  for (int y : bucket[i])
    if (y == x) return true;
  return false;
}
```

`add(x)` computes `i = x % m` and walks that bucket's chain. If some `y` equals `x`, the element is already present and the method returns `false`. Otherwise `x` is appended, `n` is incremented, and the method returns `true`. `contains(x)` does the same scan and returns `true` or `false`. Both look at a single chain.

Call	`i = x % 7`	Scanned	Returns	`n`	Non-empty buckets (after)
`add(12)`	5	—	`true`	1	5: 12
`add(25)`	4	—	`true`	2	4: 25 · 5: 12
`add(7)`	0	—	`true`	3	0: 7 · 4: 25 · 5: 12
`add(19)`	5	12	`true`	4	0: 7 · 4: 25 · 5: 12 → 19
`add(12)`	5	12 (equal)	`false`	4	unchanged

Call	`i = x % 7`	Scanned	Returns	`n`	Non-empty buckets (after)
`contains(26)`	5	12, 19	`false`	4	unchanged
`contains(19)`	5	12, 19 (equal)	`true`	4	unchanged

Table 13.5: Trace of a sequence of calls on an empty set with $m = 7$ buckets. The "Scanned" column lists, in order, the chain elements compared with `x`.

In the first three `add`s the buckets are empty, so the loop never runs. In `add(19)` it runs once ($12 \neq 19$), and 19 is appended, leaving two elements in bucket 5. In the second `add(12)`, the first comparison succeeds, the method returns `false`, and `n` stays 4.

The two `contains` calls contrast failure and success: `contains(26)` scans the whole two-element chain in vain, while `contains(19)` finds a match at the second element. Both look only at bucket 5.

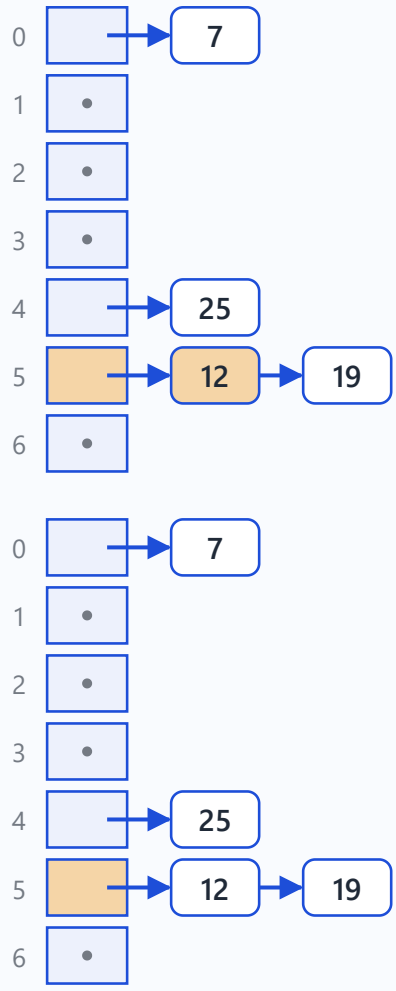


Figure 13.5: Two moments in the trace. (a) The second `add(12)`: the first element of bucket 5 is 12, so the method returns `false` and adds nothing. (b) `contains(26)`: the chain in bucket 5 (12, 19) has run out without finding 26; the other buckets are never examined.

Cost. Computing the bucket is constant time, and a chain scan is proportional to the chain's length, which averages the load factor n / m. With the load factor held constant, both methods are $O(1)$ on average. Memory is $O(n + m)$.

13.2.4 Tree-based (sorted) sets

A hash table knows nothing about order. "What is the smallest element?" or "the largest below 17?" can be answered only by scanning everything. When such questions are frequent, the set is implemented with a **balanced binary search tree**.

In a binary search tree, a node's left subtree holds smaller elements and its right subtree larger ones, so a search from the root discards half the tree at each step. A balanced tree has height $O(\log n)$, making search, insertion, and deletion $O(\log n)$. Java's `TreeSet` uses a **red-black tree**.

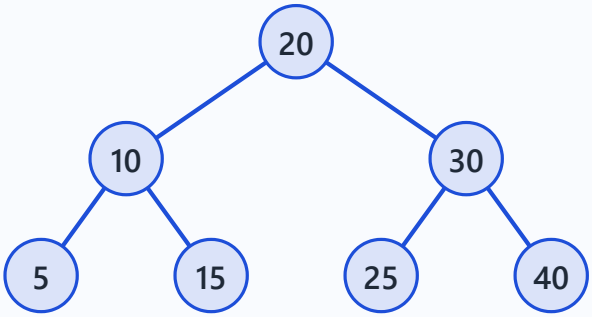


Figure 13.6: A balanced binary search tree holding the set $S = \{5, 10, 15, 20, 25, 30, 40\}$. The root is 20; the left subtree holds 10, 5, 15 and the right subtree 30, 25, 40. An inorder traversal yields the elements in sorted order.

Sorted storage has two consequences. An **inorder** traversal (left subtree, node, right subtree) yields the elements in increasing order — 5, 10, 15, 20, 25, 30, 40 for Figure 13.6. And order-based queries also take $O(\log n)$; `TreeSet` offers them through the `NavigableSet` interface.

Operation	Meaning	Result
`first()` / `last()`	smallest / largest element	5 / 40
`floor(17)`	largest element ≤ 17	15
`ceiling(17)`	smallest element ≥ 17	20
`subSet(10, 30)`	range query: $10 \le x < 30$	[10, 15, 20, 25]

Table 13.6: Extra operations of a sorted set, on $S = \{5, 10, 15, 20, 25, 30, 40\}$.

To find `floor(17)`: at the root, $20 > 17$, so go left. At 10, $10 \le 17$: 10 is a candidate; go right for a larger one. At 15, $15 \le 17$: the candidate becomes 15; go right. 15 has no right child, so the answer is 15. The search is one descent, as long as the tree's height. `ceiling(17)` follows the same path with the opposite rule and finds 20. A value in the set is its own floor and ceiling.

`subSet(10, 30)` includes the lower bound and excludes the upper, Java's usual half-open convention. A range query costs *O(log n)* to find the start plus *O(k)* for *k* results.

A tree-based set needs comparable elements: the element class implements `Comparable`, or the set is given a `Comparator`. `TreeSet` treats two elements as equal when `compare(a, b) == 0`, not by `equals`.

Common mistake

Using an ordering inconsistent with `equals`. A `TreeSet` whose `Comparator` compares people by last name alone rejects "Alice Smith" after "Ann Smith": `compare` returns 0, so the tree considers them the same element. Make the ordering agree with `equals`, for example by comparing first names after last names (`thenComparing`).

13.2.5 Union and intersection of sorted arrays

The plainest set implementation is a sorted array. Membership is a binary search, *O(log n)*; insertion and deletion shift elements, *O(n)*. In return, union and intersection of two sorted sets are very efficient, using the merge step of merge sort: one pointer per array, both advancing in a single pass. Goodrich, Tamassia, and Goldwasser call this the generic merge.

```
void unionIntersection(int[] a, int[] b,
    List<Integer> uni, List<Integer> both) {
  int i = 0, j = 0;
  while (i < a.length && j < b.length) {
    if (a[i] < b[j]) uni.add(a[i++]);
    else if (a[i] > b[j]) uni.add(b[j++]);
    else {                      // common element
      uni.add(a[i]);  both.add(a[i]);
      i++;  j++;
    }
  }
  while (i < a.length) uni.add(a[i++]);
  while (j < b.length) uni.add(b[j++]);
}
```

Pointers `i` and `j` start at 0 in `a` and `b`. While both arrays have elements left, each pass compares `a[i]` with `b[j]`:

- If `a[i] < b[j]`, `a[i]` is only in `a`: `b` is sorted, so nothing from `b[j]` on can equal it. It goes into the union and `i` advances.
- If `a[i] > b[j]`, symmetrically, `b[j]` goes into the union and `j` advances.
- If they are equal, the element is common: it goes into the union *once* and into the intersection, and both pointers advance.

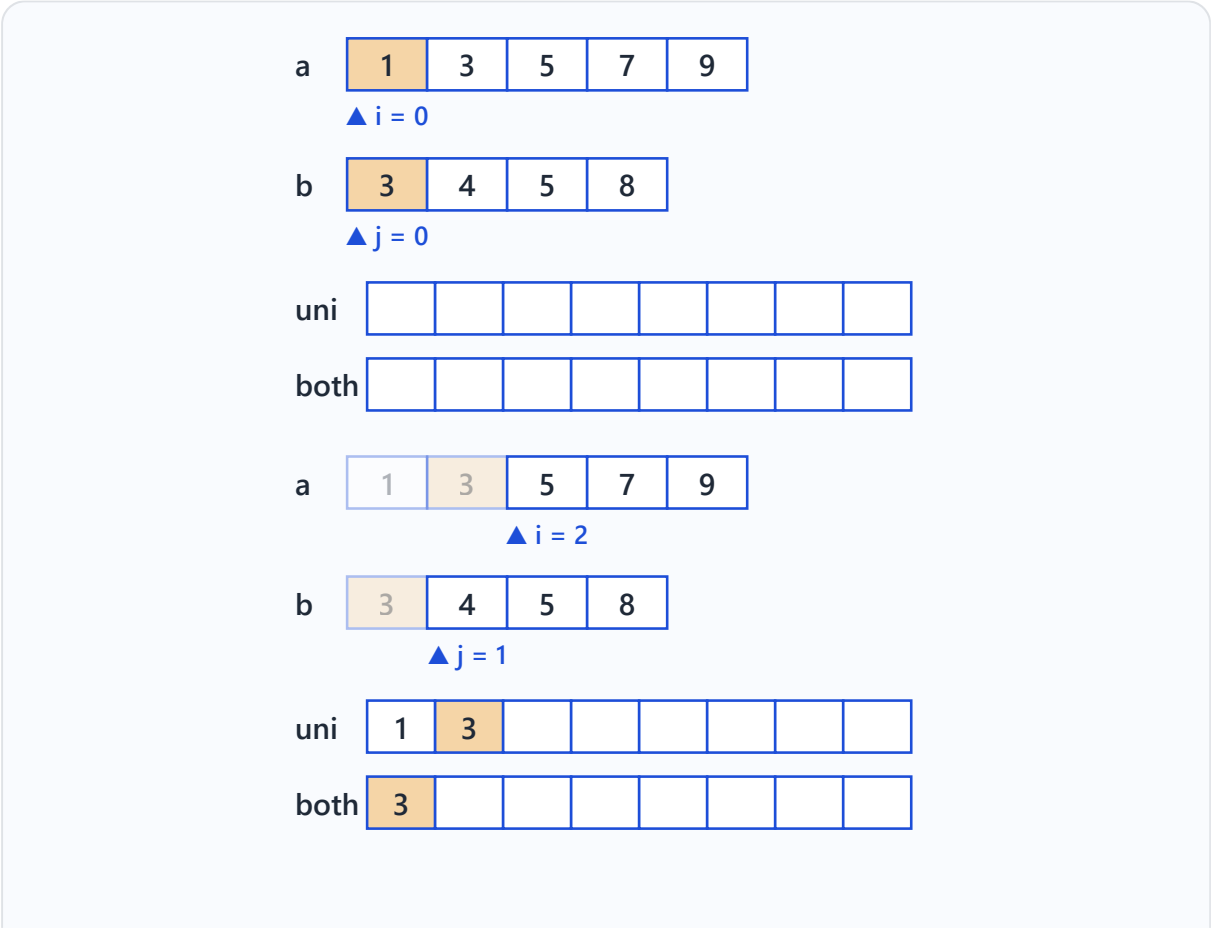
When one array runs out, the intersection is complete, and whatever is left in the other array belongs to the union alone. At most one of the two trailing loops does any work.

Let's trace the method with `a = [1, 3, 5, 7, 9]` and `b = [3, 4, 5, 8]`.

Step	Comparison	Decision	`i`	`j`	`uni`	`both`
1	1 < 3	1 only in a	1	0	[1]	[]
2	3 = 3	common	2	1	[1, 3]	[3]
3	5 > 4	4 only in b	2	2	[1, 3, 4]	[3]
4	5 = 5	common	3	3	[1, 3, 4, 5]	[3, 5]
5	7 < 8	7 only in a	4	3	[1, 3, 4, 5, 7]	[3, 5]
6	9 > 8	8 only in b	4	4	[1, 3, 4, 5, 7, 8]	[3, 5]
7	`j = 4 = b.length`	main loop ends	4	4	unchanged	[3, 5]
8	left in a: 9	added to the union	5	4	[1, 3, 4, 5, 7, 8, 9]	[3, 5]

Table 13.7: Trace of `unionIntersection([1, 3, 5, 7, 9], [3, 4, 5, 8])`. `i`, `j`, `uni`, and `both` are the values after the step.

The main loop runs six times. At step 7, `b` is exhausted and the intersection is `[3, 5]`; the first trailing loop adds the remaining 9 to the union, and the second never runs. Both outputs come out sorted, so they can serve as sorted-array sets in turn.



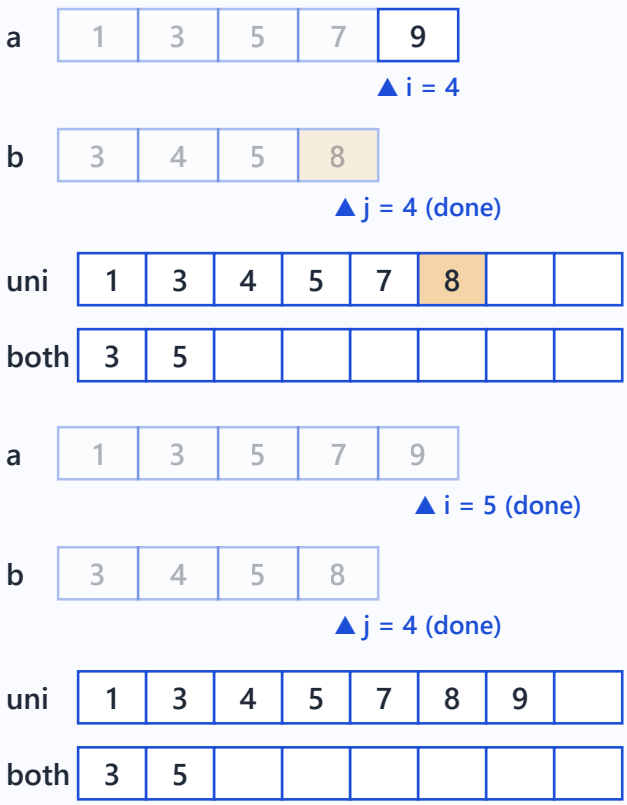


Figure 13.7: Four moments of the two-pointer union and intersection. (a) Start: `i = j = 0`. (b) The common element 3 has been added once to the union and to the intersection, and both pointers have advanced. (c) After 8 is added, `j = 4`: `b` is done. (d) The remaining 9 has been added: union [1, 3, 4, 5, 7, 8, 9], intersection [3, 5].

Cost. Every pass moves at least one pointer forward and none moves back, so with $|a| = n$ and $|b| = m$ there are at most $n + m$ steps: $O(n + m)$. Unsorted lists need a scan of the other list per element, $O(n \cdot m)$. The difference $A - B$ uses the same skeleton, keeping only the `a[i] < b[j]` elements and the leftovers of `a`.

> **Common mistake**
>
> Advancing only one pointer on a match. With only `i++`, `b[j]` still holds the same value on the next pass, and the `a[i] > b[j]` branch adds it to the union again. Both pointers must advance.

13.2.6 Comparing the implementations

Table 13.8 compares an unsorted list with the three implementations, for $|A| = n$ and $|B| = m$.

Operation	Unsorted list	Sorted array	Hash table	Balanced BST
`contains`	O(n)	O(log n)	**O(1)***	O(log n)
`add` / `remove`	O(n)†	O(n)	**O(1)***	O(log n)

Operation	Unsorted list	Sorted array	Hash table	Balanced BST
$A \cup B, A \cap B, A - B$	O(n · m)	O(n + m)	O(n + m)*	O(m log(n + m))
Sorted iteration	O(n log n)	**O(n)**	O(n log n)	**O(n)**
Minimum / `floor` / range	O(n)	O(log n)	O(n)	O(log n)

Table 13.8: Costs of set implementations. * average case (worst case O(n)). † a search is needed first to check for duplicates.

An *unsorted list* scans everything for membership, and insertion must search first to rule out a duplicate; a union searches A for each element of B, $O(n \cdot m)$. A *sorted array* searches in $O(\log n)$ but shifts elements on insertion and deletion, $O(n)$; two pointers give the set operations in $O(n + m)$. A *hash table* does single-element operations in constant average time, and a union — `contains` + `add` per element of B, plus the copy — in $O(n + m)$; but it knows no order, so sorted iteration needs a sort ($O(n \log n)$) and the minimum or a `floor` needs a full scan. A *balanced tree* does every single-element operation in $O(\log n)$; a union inserts B's m elements into a tree of at most $n + m$, $O(m \log(n + m))$, and an inorder traversal gives sorted output in $O(n)$.

In practice: **default to a hash table**, which gives the fastest membership test when order doesn't matter. **When order matters, use a tree**: everything is $O(\log n)$, and sorted queries come free. A sorted array suits a set that is built once, then only queried, with frequent unions or intersections.

13.2.7 Sets as bit vectors

If the elements come from a small range of integers, a far more compact representation exists. The set of all possible elements is the **universe**; for a universe $\{0, 1, \ldots, U - 1\}$, store U bits, with bit i set exactly when i is in the set. This is a **bit vector**.



Figure 13.8: Bit vectors over the universe 0..7. Bits are numbered 0 to 7 from right to left. For $A = \{1, 3, 4\}$, bits 1, 3, and 4 are 1, giving the value $2 + 8 + 16 = 26$. For $B = \{3, 5\}$, the value is $8 + 32 = 40$. The union `A | B` = 58 → {1, 3, 4, 5}.

With at most 32 possible elements, the set fits in one `int`, and each operation is a single instruction:

- **Insertion:** `a |= 1 << x`. `1 << x` is a **mask** with only bit x set; OR sets that bit.
- **Membership:** `(a >> x & 1) == 1`. The shift brings bit x to the right end; `& 1` isolates it.
- **Deletion:** `a &= ~(1 << x)`. The inverted mask is 1 except at bit x, so AND clears just that bit.

- **Union, intersection, difference:** `a | b`, `a & b`, `a & ~b`. Each bit is one element, so bitwise OR, AND, and NOT are exactly the set operations.
- **Size:** `Integer.bitCount(a)` counts the 1 bits.

For larger universes, `java.util.BitSet` keeps the bits in a `long` array of 64-bit words. Single-element operations stay *O(1)*; set operations go word by word, *O(U / 64)*.

Operation	`int` mask	`java.util.BitSet`
add x / is x present?	`a |= 1 << x` · `(a >> x & 1) == 1`	`set(x)` · `get(x)`
union / intersection	`a | b` · `a & b`	`or(b)` · `and(b)`
difference / size	`a & ~b` · `Integer.bitCount(a)`	`andNot(b)` · `cardinality()`

Table 13.9: Set operations on bit vectors: a single `int` mask ($U \le 32$) and `java.util.BitSet`.

A bit vector's memory depends on the universe, not on the number of elements: U bits. A million-element universe takes $10^6 / 8 = 125{,}000$ bytes, about 122 KB, against tens of megabytes for a `HashSet` of a million `Integer`s. For a large universe and a sparse set the balance flips: 3 elements from $U = 10^9$ need about 119 MB as a bit vector but a few hundred bytes in a hash set, which is then the right choice.

Common mistake

Using bits above 31 in an `int` mask. Java takes the shift distance of `1 << x` modulo 32, so `1 << 32` equals `1 << 0`: adding 32 silently adds 0. Use a `long` with `1L << x` up to 64, and a `BitSet` beyond.

13.2.8 Code trace: set algebra with bit operations

This code builds $a = \{1, 3, 4\}$ and $b = \{3, 5\}$ as bit vectors, computes their union, intersection, and difference, and tests whether 3 is in a.

```
int a = 0, b = 0;
for (int x : new int[]{1, 3, 4})
  a |= 1 << x;          // set bit x
for (int x : new int[]{3, 5})
  b |= 1 << x;
int uni = a | b;        // union
int both = a & b;       // intersection
int diff = a & ~b;      // difference
boolean has3 = (a >> 3 & 1) == 1;
```

Line	Operation	Mask / intermediate	Result (binary)	Value	Set
1	`a = 0, b = 0`	—	0000 0000	0	{}

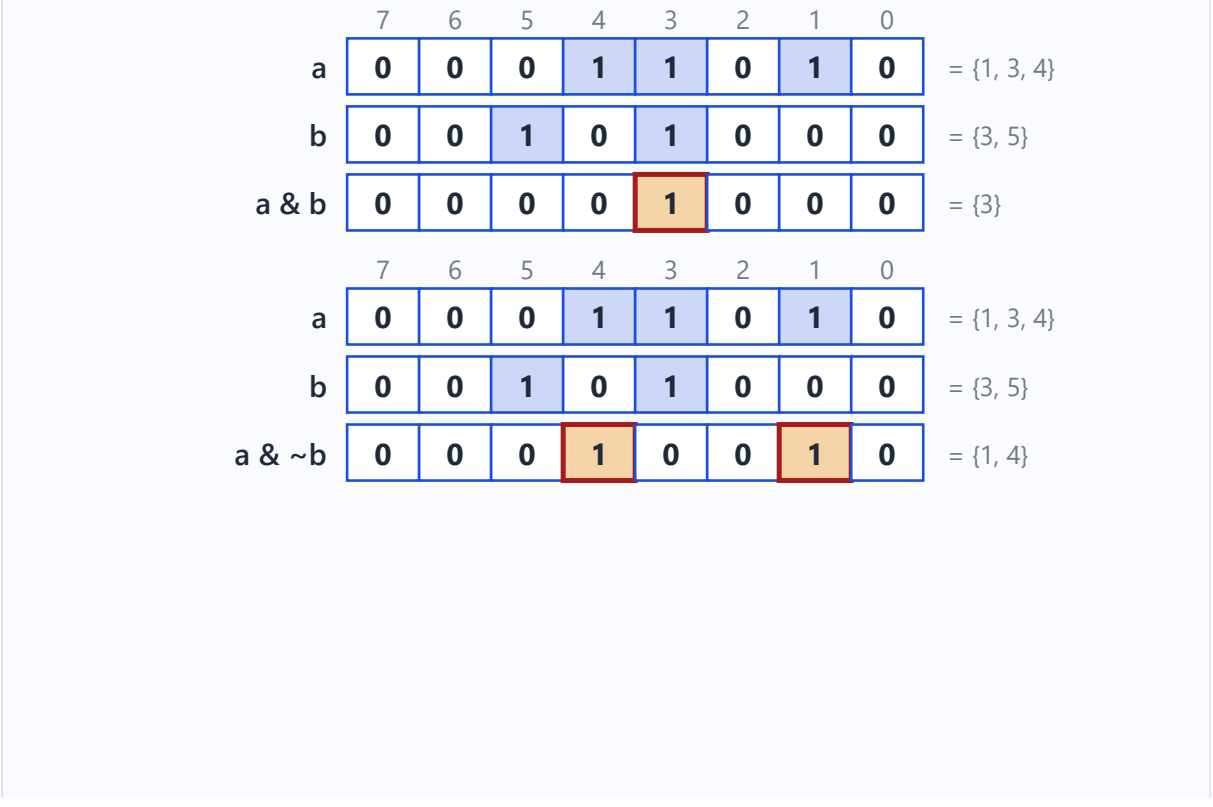
Line	Operation	Mask / intermediate	Result (binary)	Value	Set
3	`a |= 1 << 1`	0000 0010 (2)	0000 0010	2	{1}
3	`a |= 1 << 3`	0000 1000 (8)	0000 1010	10	{1, 3}
3	`a |= 1 << 4`	0001 0000 (16)	0001 1010	26	{1, 3, 4}
5	`b |= 1 << 3`	0000 1000 (8)	0000 1000	8	{3}
5	`b |= 1 << 5`	0010 0000 (32)	0010 1000	40	{3, 5}
6	`uni = a | b`	—	0011 1010	58	{1, 3, 4, 5}
7	`both = a & b`	—	0000 1000	8	{3}
8	`diff = a & ~b`	`~b` = 1101 0111 (last 8 bits)	0001 0010	18	{1, 4}
9	`(a >> 3 & 1) == 1`	`a >> 3` = 0000 0011	1	`true`	3 ∈ a

Table 13.10: Trace of the bit operations. Bits are written from 7 down to 0.

The first rows build the sets: `1 << x` is a mask with only bit *x* set (2, 8, 16), and OR adds it to `a`, which becomes 2, 10, then 26. Likewise `b = 8 + 32 = 40`.

The union keeps every bit set in either number: bits 1, 3, 4, 5, value 58, set {1, 3, 4, 5}. The intersection (AND) keeps bits set in both: only bit 3, value 8, set {3}. The difference takes two steps: `~b` flips every bit of `b`, clearing bits 3 and 5 and setting the rest, and `a & ~b` keeps the bits of `a` not in `b`, bits 1 and 4: value 18, set {1, 4}.

The last line tests membership. Shifting `a = 0001 1010` right by three gives `0000 0011`, with bit 3 at the right end; `& 1` isolates it, and the result 1 means 3 is in the set. No operation loops; each is *O(1)*.



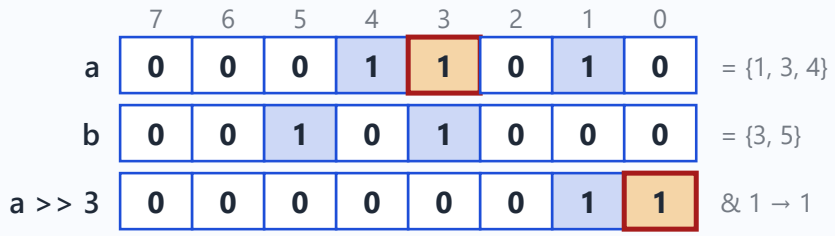


Figure 13.9: Three moments of the bit operations (top to bottom). Intersection `a & b`: only bit 3 is 1 in both numbers → {3}. Difference `a & ~b`: the bits of `a` not in `b` remain → {1, 4}. Membership: `a >> 3` brings bit 3 to the far right and `& 1` isolates it → 1, so $3 \in a$.

13.3 Multisets and multimaps

Forbidding duplicates usually helps, but sometimes duplicates are the point. How often does "and" occur in a text? How many apples are in the basket? How many votes did a candidate get? Counting questions like these call for an extension of the set.

13.3.1 Multisets

Definition 13.2

A **multiset** (or **bag**) is a set in which an element may occur more than once. Order is still irrelevant; what matters is how many times each element occurs, its **multiplicity**.

In *M = {a, a, b, c, c, c}*, a has multiplicity 2, b 1, and c 3. Writing each distinct element once with its multiplicity, *{a:2, b:1, c:3}*, carries the same information. That is the natural implementation: an **element → counter** map, which with a hash table reduces every operation to one key lookup.

Operation	Meaning	With hashing
`add(e)` / `add(e, n)`	add e once / n times (counter += n)	O(1)*
`count(e)`	how many times is e present? (0 if absent)	O(1)*
`remove(e)` / `remove(e, n)`	decrement the counter; delete the entry if it reaches 0	O(1)*
`size()`	number of elements including repeats (6 for *M*)	O(1)

Table 13.11: The multiset ADT (Goodrich, Tamassia, and Goldwasser) and its costs with a hash-based map. * average case.

`size()` counts repeats: *M* has 3 *distinct* elements but size 6. Rather than summing the counters each time, the implementation keeps a counter `n`, updated by every insertion and removal, so `size()` is *O(1)*.

Word frequencies, shopping-cart items, and vote counts are classic multisets. Whether two words are anagrams can be tested by comparing their letter multisets.

13.3.2 A multiset in Java

Java's library has no multiset class, but one is easy to build on a `HashMap<E, Integer>`:

```
class Multiset<E> {
    private final Map<E, Integer> counts = new HashMap<>();
    private int n = 0;                                  // number of elements, repeats
included
    void add(E e) { counts.merge(e, 1, Integer::sum); n++; }  // 1 if absent, else +1
    int count(E e) { return counts.getOrDefault(e, 0); }
    boolean remove(E e) {
        Integer c = counts.get(e);
        if (c == null) return false;                    // e is absent
        if (c == 1) counts.remove(e);                   // counter would hit 0: delete
entry
        else counts.put(e, c - 1);
        n--;
        return true;
    }
    int size() { return n; }
}
```

`counts` maps each distinct element to its counter; `n` counts all elements, repeats included.

- `add(e)` calls `counts.merge(e, 1, Integer::sum)`, which inserts `e` with value 1 if absent and otherwise combines old and new values with the given function — addition, so the counter goes up by one. Then `n` is incremented.
- `count(e)` uses `getOrDefault(e, 0)`, returning 0 rather than `null` for an absent element — which, in a multiset, does occur 0 times.
- `remove(e)` reads the counter. If it is `null`, the element is absent and the method returns `false`. A counter of 1 would drop to 0, so the entry is deleted; a larger counter is decremented. Either way `n` drops and the method returns `true`.
- `size()` simply returns `n`.

After `add("a"); add("a");`, `count("a")` returns 2, whereas a `Set` would ignore the second call and record only that "a" is present. For sorted iteration, use a `TreeMap<E, Integer>`: operations rise to *O(log n)*, but elements come out in order.

> **Common mistake**
>
> Leaving entries with a zero counter in the map. If `remove` always writes `counts.put(e, c - 1)`, zero entries pile up, and the `c == null` check no longer detects absence: removing such an element again makes its counter −1, returns `true`, and wrongly decrements `n`. A multiset never stores a zero counter.

13.3.3 Code trace: Multiset counters

We trace these calls on an empty `Multiset<String>`: `add("a")`, `add("b")`, `add("a")`, `add("c")`, `count("a")`, `remove("a")`, `remove("b")`, `remove("x")`, `count("b")`. The method bodies are:

```
void add(E e) {
  counts.merge(e, 1, Integer::sum);  n++;
}
int count(E e) {
  return counts.getOrDefault(e, 0);
}
boolean remove(E e) {
  Integer c = counts.get(e);
  if (c == null) return false;
  if (c == 1) counts.remove(e);
  else counts.put(e, c - 1);
  n--;
  return true;
}
```

Call	What happens	Returns	`counts`	`n`
`add("a")`	"a" absent → inserted with counter 1	—	{a:1}	1
`add("b")`	"b" absent → counter 1	—	{a:1, b:1}	2
`add("a")`	"a" present → *1 + 1 = 2*	—	{a:2, b:1}	3
`add("c")`	"c" absent → counter 1	—	{a:2, b:1, c:1}	4
`count("a")`	`getOrDefault` → 2	2	unchanged	4
`remove("a")`	`c = 2 > 1` → counter drops to 1	`true`	{a:1, b:1, c:1}	3
`remove("b")`	`c == 1` → entry deleted	`true`	{a:1, c:1}	2
`remove("x")`	`c == null` → element absent	`false`	unchanged	2
`count("b")`	"b" not in the map → 0	0	unchanged	2

Table 13.12: Trace of `Multiset`. `counts` and `n` are the values after the call.

The third row shows the difference from a set: `merge` adds 1 to the old counter, making it 2, and `n` rises to 3, where a `Set` would not have changed. The two successful `remove`s take different branches: "a" has counter 2, so the `else` branch lowers it to 1 and the entry stays; "b" has counter 1, so its entry is deleted. For the absent "x", `counts.get` returns `null` and the method returns `false` at once. Finally, `count("b")` returns 0: a removed element looks like one never added. The result is $M = \{a, c\}$, `n = 2`.

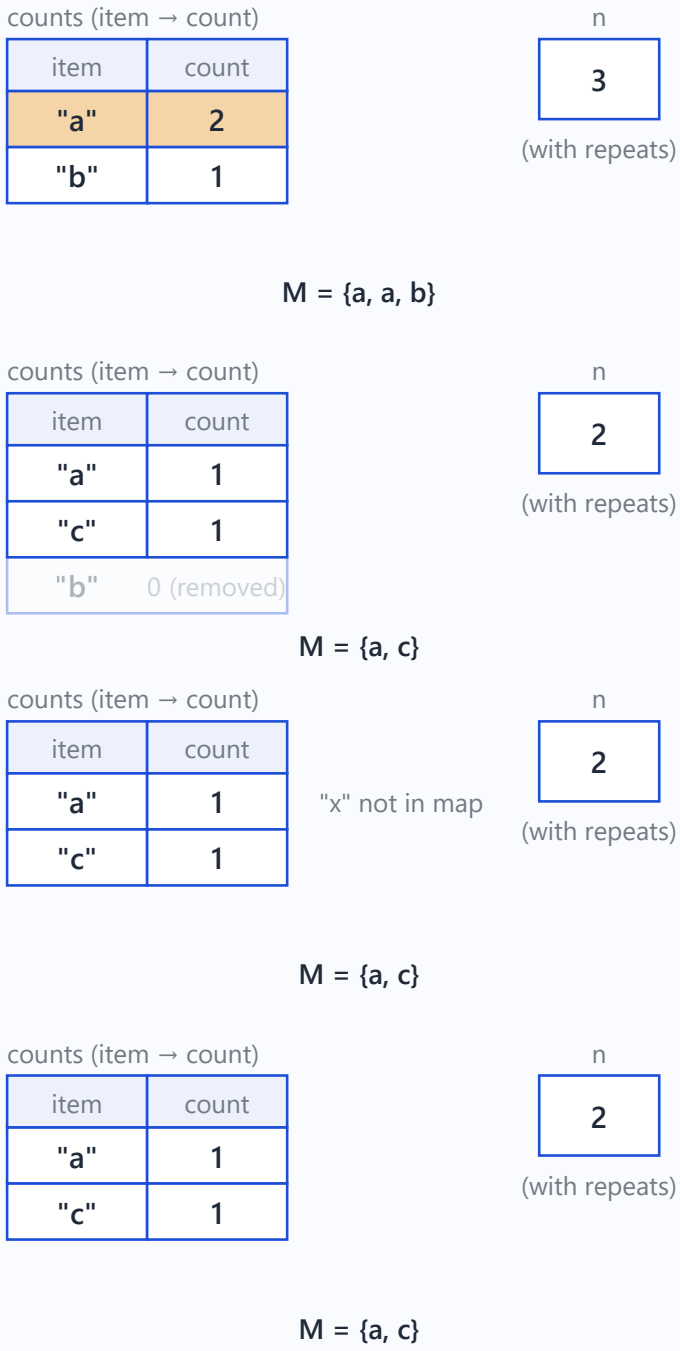


Figure 13.10: Four moments in the trace of `Multiset`. (a) "a" added a second time: counter 2, `n = 3`. (b) `remove("b")`: the counter was 1, so the entry was deleted (zero counters are not stored), `n = 2`. (c) `remove("x")`: "x" is not in the map, and the call returns `false`. (d) The result, *M = {a, c}*, `n = 2`.

Cost. Each method touches the map once (`remove` at most twice), *O(1)* on average with hashing. Memory is *O(d)* for *d* distinct elements: a word occurring millions of times still takes one entry.

Example 13.2

The letters of "kayak" are added in order to a `Multiset<Character>`. What do `count('a')` and `size()` return?

1. **k:** absent, counter 1. Map {k:1}, `n = 1`.
2. **a:** absent, counter 1. {k:1, a:1}, `n = 2`.
3. **y:** absent. {k:1, a:1, y:1}, `n = 3`.
4. **a:** present, *1 + 1 = 2*. {k:1, a:2, y:1}, `n = 4`.
5. **k:** present, counter 2. {k:2, a:2, y:1}, `n = 5`.
6. **Result.** `count('a')` returns 2, `size()` 5. The number of distinct letters is the map's size, 3.

13.3.4 Multimaps

A multiset records *how many times* an element occurs. Sometimes we need the repeats themselves: to know *which* students take a course, each course key must map to a list of students.

Definition 13.3

A **multimap** is a map in which a key may have several values. In the ADT of Goodrich, Tamassia, and Goldwasser, `get(k)` returns all values of a key, `put(k, v)` adds a (key, value) pair, `remove(k, v)` deletes one pair, `removeAll(k)` all pairs of a key, and `size()` counts all pairs.

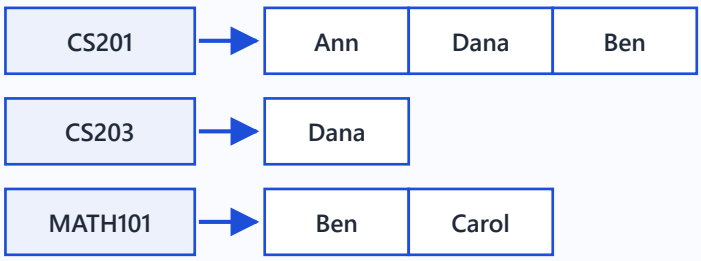


Figure 13.11: A course → students multimap. CS201 maps to three students (Ann, Dana, Ben), CS203 to one (Dana), and MATH101 to two (Ben, Carol). Dana and Ben appear in the value lists of more than one key.

Java has no multimap class either; a `Map<K, List<V>>` mapping each key to a list of values does the job:

```
Map<String, List<String>> m = new HashMap<>();
void put(String k, String v) {   // create an empty list if absent
  m.computeIfAbsent(k, x -> new ArrayList<>()).add(v);
}
List<String> get(String k) {
  return m.getOrDefault(k, List.of());
}
```

In `put`, `computeIfAbsent(k, x -> new ArrayList<>())` returns the key's list, first creating and storing an empty one if the key is absent; `add(v)` then appends to it. For a missing key, `get` returns an unmodifiable empty list (`List.of()`) instead of `null`, sparing the caller a `null` check.

The map in Figure 13.11 takes six `put`s: (CS201, Ann), (CS201, Dana), (CS201, Ben), (CS203, Dana), (MATH101, Ben), (MATH101, Carol). `get("CS201")` returns `[Ann, Dana, Ben]` and `get("PHYS101")` an empty list. The size is the number of pairs, 6; there are 3 keys.

Cost. With hashing, `put` and `get` are *O(1)* on average (`get` returns the list without copying). `remove(k, v)` finds the list in *O(1)*, then searches it: *O(1 + list length)*. In short, a multiset keeps counts; a multimap keeps the values themselves.

13.4 Disjoint sets and Union-Find

So far we have worked with one set, or two. Some problems involve many sets that merge over time: groups of friends in a social network, cities linked by roads, adjacent same-colored pixels in an image. They all ask "Are these two items in the same group?" and "Merge these two groups." This section builds a structure that answers both very quickly.

13.4.1 Disjoint sets

Definition 13.4

Disjoint sets are a collection of sets no two of which share an element, so every item belongs to exactly *one* set. The structure that holds them is **Union-Find** (or Disjoint Set Union, DSU), with two operations:

- `find(x)` returns the **representative** of the set containing x. Two items are in the same set if and only if `find(x) == find(y)`.
- `union(x, y)` merges the set of x and the set of y into a single set. For example, merging *{1, 2}* and *{3}* gives *{1, 2, 3}*.

The representative is the set's "name." Which element it is doesn't matter, as long as `find` returns the *same* one for every element of the set; then "same set?" takes two `find`s and a comparison.

Union-Find stores each set as a **tree** in which each node points only to its **parent**, so arrows run from child to ancestor. The root points to itself and represents the set. Several trees form a **forest**.

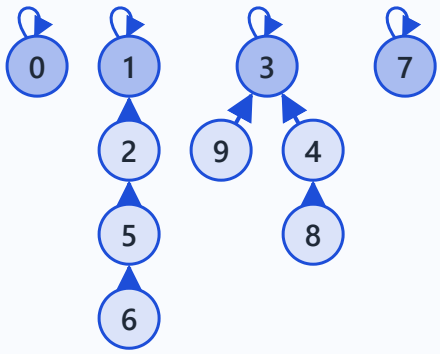


Figure 13.12: Four disjoint sets as trees. {0} and {7} are single-node sets. {1, 2, 5, 6} is a chain: 6 → 5 → 2 → 1. In {3, 9, 4, 8}, 9 and 4 point to 3, and 8 points to 4. Each root points to itself (the loop arrow) and represents its set.

In Figure 13.12, `find(6)` follows 6 → 5 → 2 → 1 and stops at 1, which points to itself. `find(8)` follows 8 → 4 → 3 and returns 3. Since $1 \neq 3$, 6 and 8 are in different sets. Merging the sets takes one new arrow: point one root at the other, say 1 at 3.

These are not search trees: nodes have no smaller–larger order and any number of children. A tree's only job is to connect the elements of a set to a common root.

13.4.2 The magnet example

Picture 14 strong magnets on a table. Two magnets that touch stick together for good; stuck magnets form a group, and a magnet touching any member of another group merges the two groups. As merging goes on, we want to know at any moment: "Are these two magnets in the same group?"

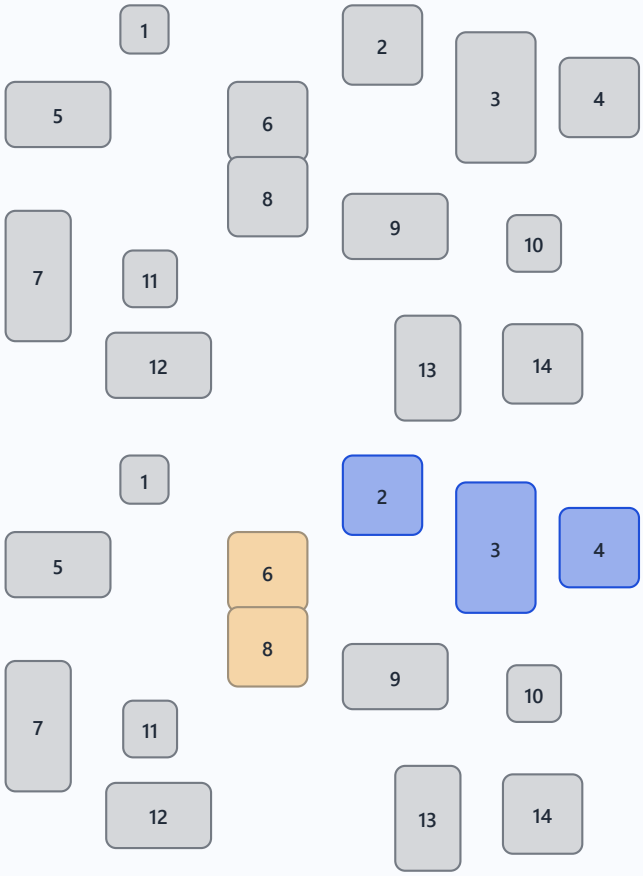
Number the magnets 1 to 14. Initially each is in its own group: 14 disjoint sets, with `find(x) = x` for every x. A touch is a `union`; "same group?" compares two `find`s.

Merge	Groups (after)	Number of groups
start	every magnet on its own	14
`union(6, 8)`	{6, 8}	13
`union(3, 4)`	{6, 8} · {3, 4}	12
`union(2, 3)`	{6, 8} · {2, 3, 4}	11
`union(9, 10)`, `union(13, 14)`, `union(9, 13)`	{6, 8} · {2, 3, 4} · {9, 10, 13, 14}	8
`union(5, 7)`, `union(7, 11)`, `union(11, 12)`	{6, 8} · {2, 3, 4} · {9, 10, 13, 14} · {5, 7, 11, 12}	5
`union(4, 10)`	{6, 8} · {2, 3, 4, 9, 10, 13, 14} · {5, 7, 11, 12}	4

Table 13.13: Merges and groups in the magnet example. Magnets still on their own are not listed.

Each successful merge reduces the number of groups by exactly one. In `union(2, 3)`, the lone 2 joins 3's group {3, 4}, making {2, 3, 4}. Three merges then build {9, 10, 13, 14}, and three more {5, 7, 11, 12}, leaving five groups: four larger ones and 1 on its own.

Are 4 and 10 in the same group? `find(4)` returns the representative of {2, 3, 4} and `find(10)` that of {9, 10, 13, 14}; they differ, so no. `union(4, 10)` then merges the two groups. Eventually all magnets end up in one group with a single root.



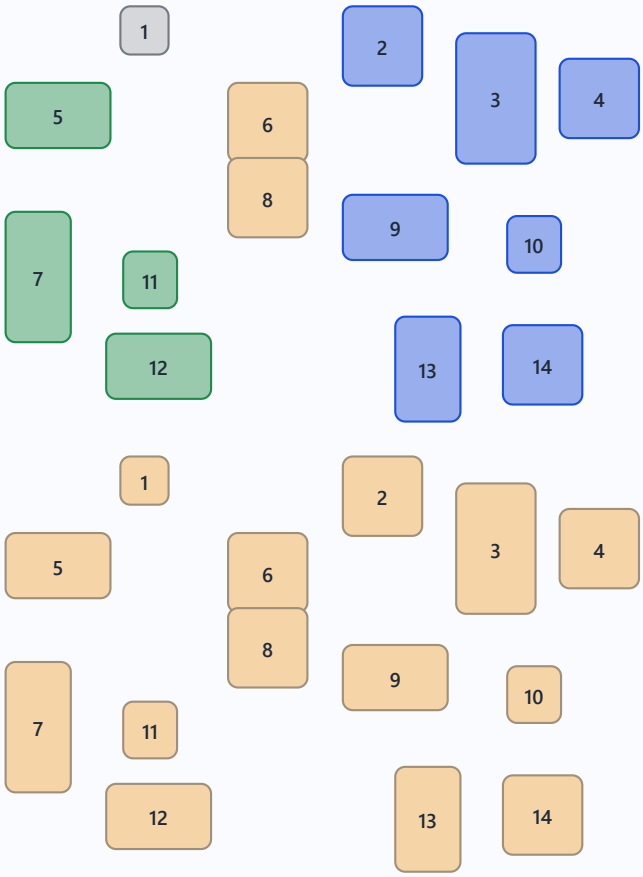


Figure 13.13: Four moments of the magnet example; magnets of the same color are in the same group. (a) 14 magnets, each in its own group. (b) {6, 8} and {2, 3, 4} have formed. (c) `union(4, 10)` has merged two groups. (d) All the magnets are in one group.

Note that Union-Find remembers only who is in which group, not *how* the groups formed: whether 2 touched 3 or 4 cannot be asked afterward. Discarding that history is one reason the structure is so fast.

13.4.3 The find and union operations

More precisely:

`find(x)` returns the name of x's set, the root of its tree, by following parent links up to the root. Unlike a search-tree lookup, which seeks a *value* by descending from the root, `find` starts from a known node and climbs.

`union(x, y)` replaces two sets with their union by attaching one root below the other. The sets are disjoint, so there is no duplicate check and no copying: a single parent link changes. Set union in Section 13.1 took $O(n + m)$; here, once the roots are known, it is $O(1)$.

In pseudocode:

```
find(x):
  if x.parent != x:
    x.parent = find(x.parent)  // path compression
  return x.parent

union(x, y):
  root1 = find(x)
  root2 = find(y)
  if root1 != root2:
    // make the root of one set
    // a child of the other's root
    root1.parent = root2
```

`find` is recursive: a node that is not its own parent finds its parent's root. The assignment `x.parent = find(x.parent)` also makes that root x's new parent; this is **path compression**, covered in Section 13.5.4. `union` finds both roots and, if they differ, makes the second the parent of the first; if they are equal, the items already share a set.

Common mistake

Linking the items instead of their roots. `parent[x] = y` in `union(x, y)` loses x's old parent link if x is not a root, splitting its tree: x and its subtree move to y's set while x's former ancestors stay behind. Link the roots: `parent[find(x)] = find(y)`.

13.4.4 Mapping objects to indices

Union-Find is most efficient on an integer array: the elements are *0..n − 1*, and `parent[i]` holds the index of *i*'s parent, so moving up is a single array read.

Real elements are often letters, cities, or people, so each object first gets an index — any index, but fixed once assigned — through an object → index hash map. Then `parent[i] = i` makes every element the root of its own one-element set; this initialization is `makeSet`.

Letter	E	F	I	D	C	A	J	L	G	K	B	H
Index	0	1	2	3	4	5	6	7	8	9	10	11

Table 13.14: Mapping twelve letters to indices.

```
Map<Character, Integer> index = new HashMap<>();   // 'E'→0, 'F'→1, …
int[] parent = new int[12];
for (int i = 0; i < 12; i++) parent[i] = i;              // 12 singleton sets
```

The first line holds the letter → index map; the next two build a 12-cell parent array with each cell pointing to itself: 12 singleton sets. A call like `union('C', 'K')` looks up the indices (4 and 9) and then works on the array. The map is *O(1)* on average, so the translation adds nothing to the cost; initialization is *O(n)*.

13.4.5 union step by step

Let's trace `union` calls on the 12 letters of Table 13.14, without path compression, under this rule: **attach the root of the smaller tree to the root of the larger; on equal sizes, the second root goes under the first.** Each root stores its tree's size (number of elements). Section 13.5.2 explains the rule; here we just apply it.

Call	Roots	Sizes	Linked	Cell changed
`union(C, K)`	C, K	1, 1 (equal)	K → C	`parent[9] = 4`
`union(F, E)`	F, E	1, 1 (equal)	E → F	`parent[0] = 1`
`union(A, J)`	A, J	1, 1 (equal)	J → A	`parent[6] = 5`
`union(A, B)`	A, B	2, 1	B → A	`parent[10] = 5`
`union(C, D)`	C, D	2, 1	D → C	`parent[3] = 4`
`union(D, I)`	C, I	3, 1	I → C	`parent[2] = 4`
`union(L, F)`	L, F	1, 2	L → F	`parent[7] = 1`
`union(C, A)`	C, A	4, 3	A → C	`parent[5] = 4`
`union(A, B)`	C, C	—	already the same set	none
`union(H, G)`	H, G	1, 1 (equal)	G → H	`parent[8] = 11`
`union(H, F)`	H, F	2, 3	H → F	`parent[11] = 1`
`union(H, B)`	F, C	5, 7	F → C	`parent[1] = 4`

Table 13.15: Trace of union by size (no path compression). The "Roots" column gives the two `find` results; "Sizes" gives the sizes of those roots' trees.

In `union(D, I)`, D is not a root: `find(D)` returns its parent C, and the singleton I joins C's tree of size 3. In `union(C, A)`, C's tree {C, K, D, I} (size 4) meets A's {A, J, B} (size 3); the smaller goes under C, and the new size is 7.

In the second `union(A, B)`, both `find`s return C (B's path is B → A → C), so nothing changes: merging two elements of one set is harmless. In `union(H, B)`, H's root is F and B's is C; F's tree {F, E, L, H, G} (size 5) goes under C's (size 7). All 12 elements now form one set with root C.

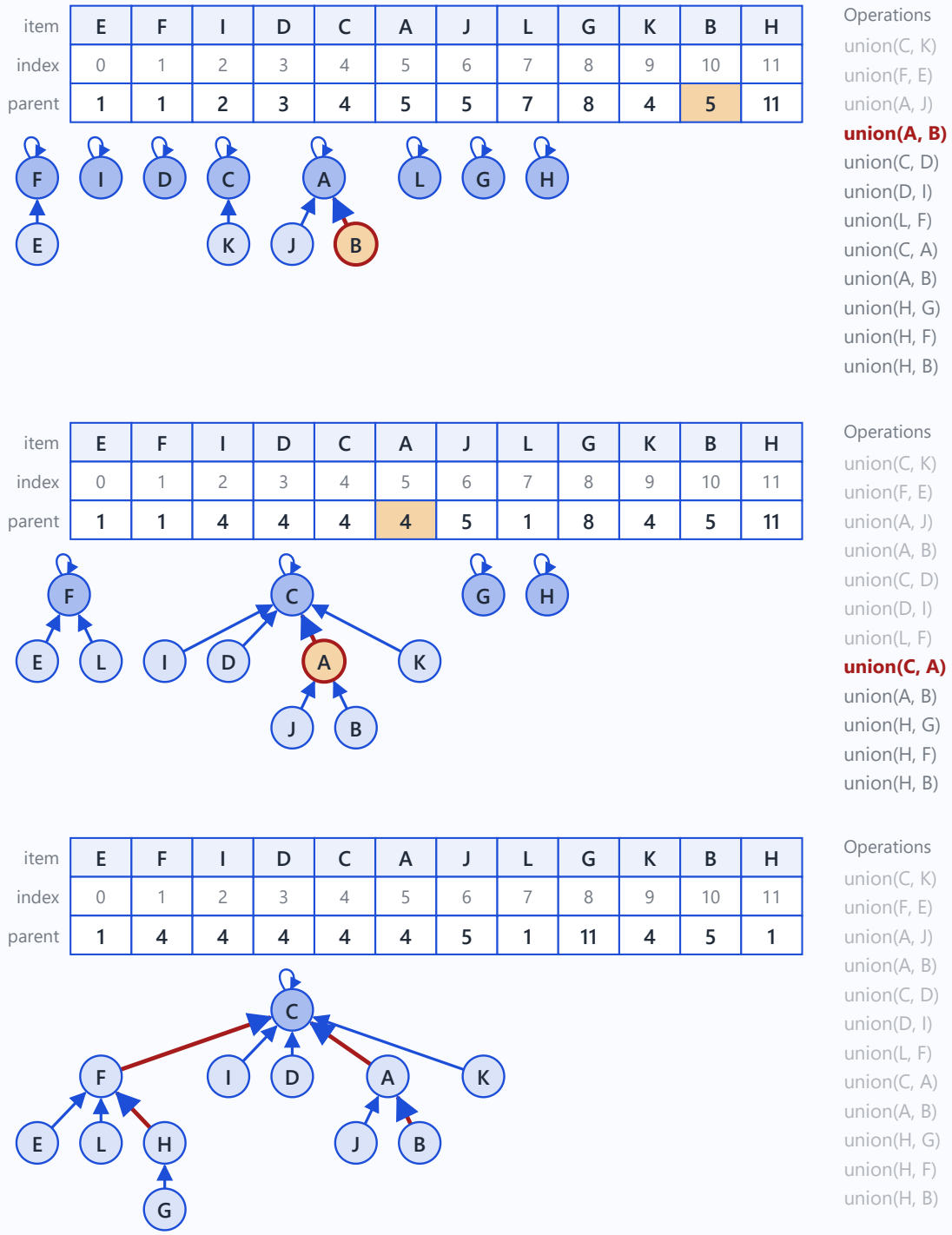


Figure 13.14: Three moments of the union-by-size trace (top to bottom). Left, the table of letters, indices, and `parent`; middle, the list of operations; right, the forest. (a) After `union(A, B)`: B is under A. (b) After `union(C, A)`: A's tree is attached to C, size 7. (c) The result: one set, root C.

Are H and B now in the same set? `find(H)` takes two steps (H → F → C), `find(B)` two (B → A → C); the roots match, so yes, after 4 steps. The tree's height is 3, the deepest node being G (G → H → F → C). Such a small height for 12 elements is no accident; it comes from union by size.

13.4.6 Another idea: Quick-Find

A simpler representation uses no trees. In Sedgewick and Wayne's Quick-Find, a flat array `id` holds each element's set name, equal for elements of the same set, so `find(p)` is a single read, `id[p]`: *O(1)*.

`union` pays: `union(p, q)` must rename *every* element of p's set, and since their positions are unknown, it scans the whole array: *O(n)*.

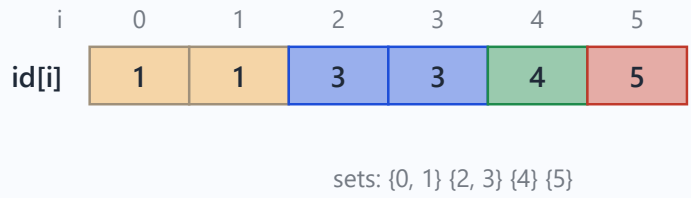


Figure 13.15: The Quick-Find `id` array after `union(0, 1)` and `union(2, 3)`: `id = [1, 1, 3, 3, 4, 5]`. There are four sets: {0, 1}, {2, 3}, {4}, {5}. Cells of the same color are in the same set.

Method	`find`	`union`	Total for $n-1$ `union`s
Quick-Find (array of set names)	O(1)	O(n)	$O(n^2)$
Quick-Union (parent tree, naive)	O(h), worst O(n)	O(h)	worst $O(n^2)$
Rank/size + path compression	O(α(n)) amortized		≈ O(n)

Table 13.16: Costs of three Union-Find methods (following Sedgewick and Wayne's classification). *h*: tree height.

The table maps out this section and the next. Quick-Find makes `find` fast, but gathering n elements into one set takes $n-1$ array-scanning `union`s, $O(n^2)$. The naive parent tree (Quick-Union) makes `union` cheap, but trees can grow tall, and `find` depends on height. The best result combines two improvements that keep trees flat; the $\alpha(n)$ in the table, introduced in Section 13.5.5, never exceeds 4 in practice.

13.4.7 Code trace: Quick-Find union

```
int find(int p) { return id[p]; }  // O(1)
void union(int p, int q) {
  int pid = id[p], qid = id[q];
  if (pid == qid) return;
  for (int i = 0; i < id.length; i++)
    if (id[i] == pid)
      id[i] = qid;           // O(n) scan
}
```

`union` reads the two set names; if they are equal, it returns at once. Otherwise it writes `qid` into every cell whose name is `pid`.

Starting from Figure 13.15, `id = [1, 1, 3, 3, 4, 5]`, we call `union(1, 3)` and then `union(4, 3)`.

Call	`pid`, `qid`	`i`	`id[i] == pid`?	`id` (after)
`union(1, 3)`	1, 3	0	1 = 1 yes → 3	[3, 1, 3, 3, 4, 5]
		1	1 = 1 yes → 3	[3, 3, 3, 3, 4, 5]
		2	3 ≠ 1	unchanged
		3	3 ≠ 1	unchanged
		4	4 ≠ 1	unchanged
		5	5 ≠ 1	unchanged
`union(4, 3)`	4, 3	0	3 ≠ 4	unchanged
		1	3 ≠ 4	unchanged
		2	3 ≠ 4	unchanged
		3	3 ≠ 4	unchanged
		4	4 = 4 yes → 3	[3, 3, 3, 3, 3, 5]
		5	5 ≠ 4	unchanged

Table 13.17: Trace of Quick-Find `union`. Each row is one pass of the loop.

`union(1, 3)` renames cells 0 and 1; `union(4, 3)` renames only cell 4. Yet each call examines all six cells.

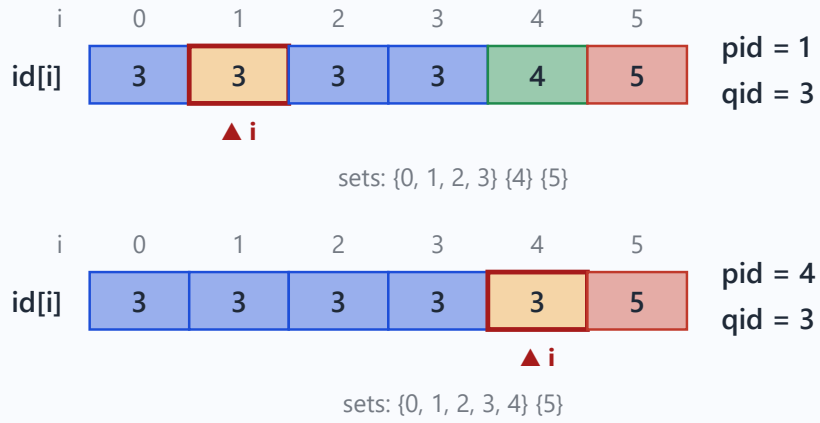


Figure 13.16: Two moments of the Quick-Find trace. Top, the second pass of `union(1, 3)`: 0 and 1 have been renamed 3, giving the sets {0, 1, 2, 3}, {4}, {5}. Bottom, the fifth pass of `union(4, 3)`: 4 has been renamed 3, giving {0, 1, 2, 3, 4}, {5}.

Cost. Every `union` examines n cells, so the $n - 1$ merges that gather everything into one set cost $(n - 1) \cdot n$ visits, $O(n^2)$: 56 for 8 elements, about 10^{12} for a million — useless in practice. Making `find` constant makes `union` expensive; the parent tree reverses the balance.

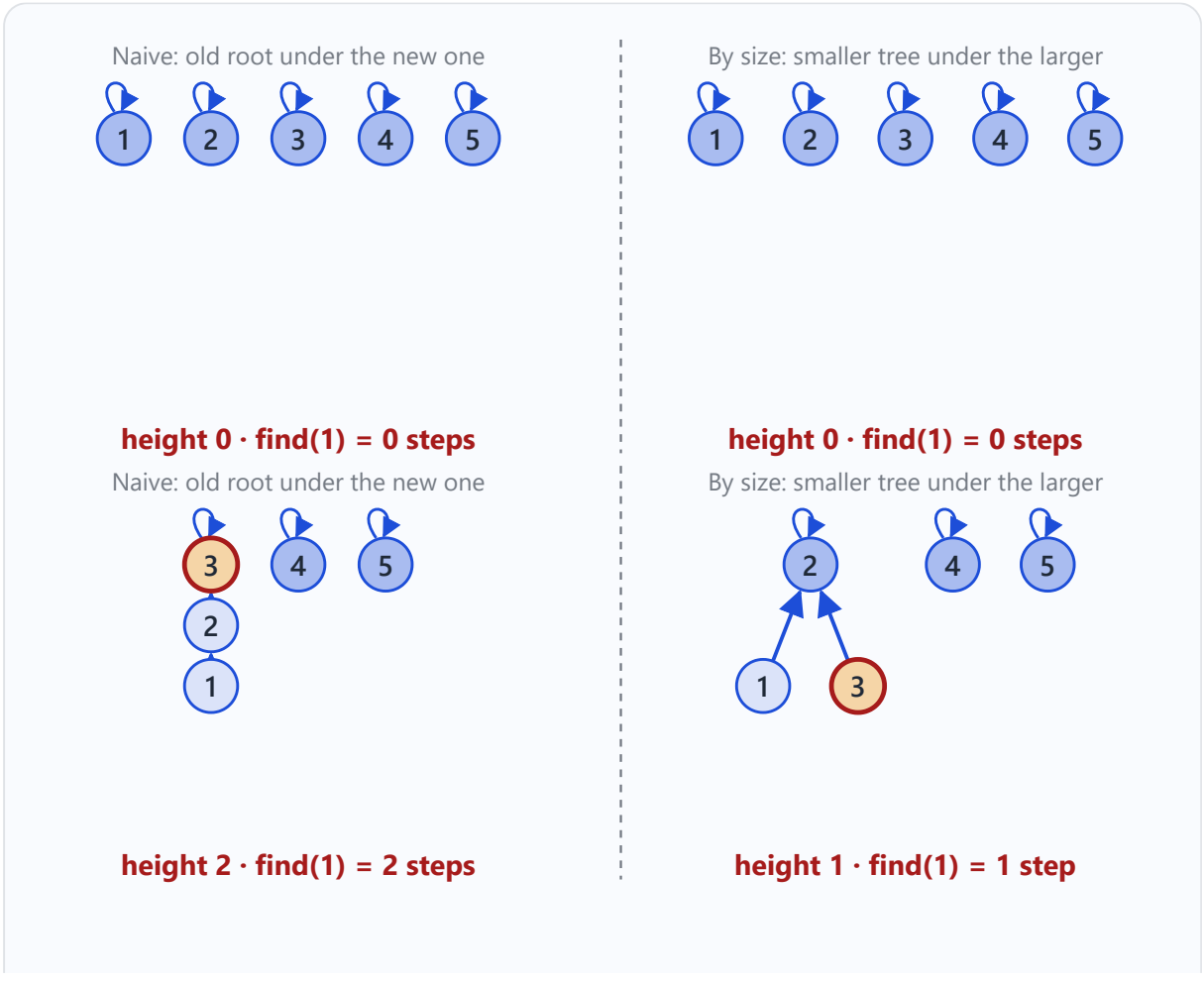
13.5 Making Union-Find fast

With a parent tree, `union` is one assignment once the roots are known. All the cost is in `find`, which takes as many steps as the node's distance from the root, so speeding up Union-Find means keeping trees *flat*. This section presents two simple ideas and the surprising result of combining them.

13.5.1 Why improve it?

Let the naive `union(x, y)` always put `x`'s root below `y`'s, and call `union(1, 2)`, `union(2, 3)`, `union(3, 4)`, `union(4, 5)` on five singletons. Each call puts the growing tree under a single node, lengthening the chain by one, until the tree is 1 → 2 → 3 → 4 → 5, of height $n - 1 = 4$; `find(1)` follows four links.

Under "attach the smaller tree under the larger," 1 still goes under 2, but then each singleton (3, 4, 5) is added to the root 2. The result is a star of height 1, and every `find` takes at most one step.



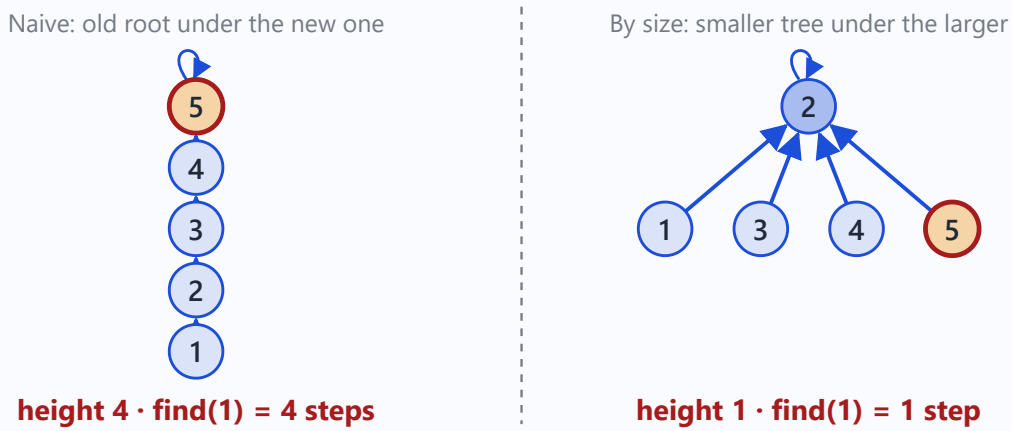


Figure 13.17: Naive union (left) and union by size (right) on the same sequence of calls. Top, the start: five singleton sets. Middle, after two `union`s: the naive tree is the chain $1 \to 2 \to 3$ (height 2), the by-size tree a star (height 1). Bottom, after four `union`s: the naive tree is a chain of height 4, while the by-size tree still has height 1.

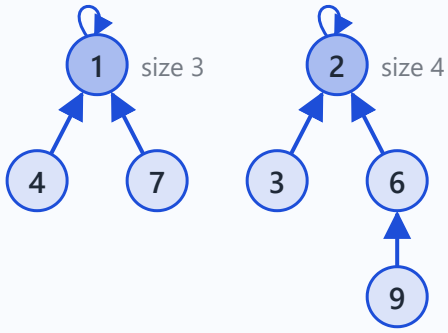
On a chain, every `find` takes $O(n)$, so a badly ordered sequence of m `find`s costs $O(m \cdot n)$, possibly worse than Quick-Find. Two simple ideas keep trees flat: (1) **union by size or by rank** and (2) **path compression**. Together they make each operation nearly constant: $O(\alpha(n))$.

13.5.2 Union by size

Definition 13.5

Union by size: each root stores the **size** (number of elements) of its tree. The root of the smaller tree is attached to the root of the larger, whose size becomes the sum of the two. Equal sizes are linked by a fixed rule (for example, the second root under the first).

The reasoning: attaching a tree below another root deepens *every* node of that tree by one, so moving the smaller tree deepens fewer nodes, and the larger tree's nodes keep their depths.



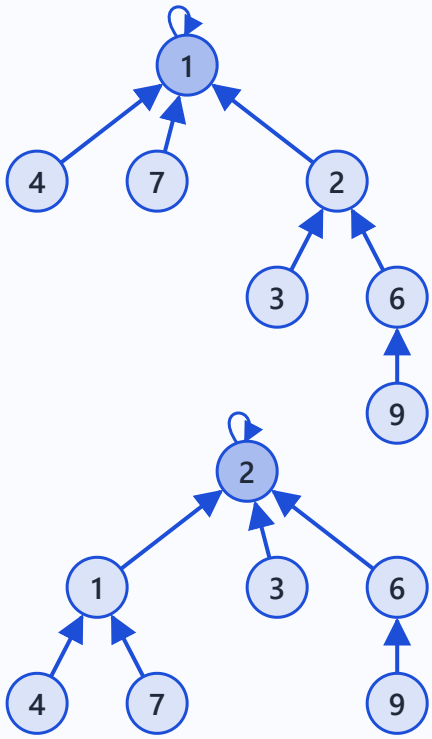


Figure 13.18: The effect of union by size. Top, two sets: {1, 4, 7} with root 1 (size 3, height 1) and {2, 3, 6, 9} with root 2 (size 4, height 2; 9 is under 6). Bottom left, naive union: 2, the root of the larger tree, goes under 1, the root of the smaller; the height rises to 3. Bottom right, union by size: 1 goes under 2; the height stays 2, and `size[2] = 3 + 4 = 7`.

In Figure 13.18, putting the larger tree under the smaller deepens 2, 3, 6, and 9, and 9 sinks from depth 2 to 3. Moving the smaller tree deepens only the shallow nodes 1, 4, and 7; 4 and 7 reach depth 2, 9 stays deepest, and the height remains 2.

Theorem 13.1 (Height under union by size)

Under union by size, with or without path compression, no node in a forest of n elements has depth greater than $\lfloor \log_2 n \rfloor$, so `find` and `union` take $O(\log n)$ in the worst case.

Proof. The depth of a node x grows only when its tree is attached below another root, which by the rule happens only if its tree is no larger than the other. The tree containing x therefore at least doubles, from size s to at least $2s$. Starting from 1 and never exceeding n, the size can double at most $\lfloor \log_2 n \rfloor$ times, and so can x's depth grow. Path compression only reduces depths. □

This matches Section 13.4.5: $\lfloor \log_2 12 \rfloor = 3$, and the height was 3. The bound is tight: merging 8 elements into equal trees of two, then four, then eight gives height $\log_2 8 = 3$, and nothing worse is possible.

13.5.3 Union by rank

Tracking height instead of size works as well, but under path compression the true height is costly to maintain: a `find` may lower it, and detecting that means walking the whole tree. So we keep an upper bound instead.

> **Definition 13.6**
>
> The **rank** is an integer stored at each root that bounds its tree's height from above; a new one-element set has rank 0. In **union by rank**, the root of lower rank is attached to the root of higher rank, and ranks don't change. On equal ranks, the roots are linked and the new root's rank goes up by one.

CLRS splits the operations into three parts:

```
makeSet(x):
  parent[x] = x
  rank[x] = 0

find(x):   // with path compression
  if x != parent[x]:
    parent[x] = find(parent[x])
  return parent[x]

link(x, y):   // x, y are roots
  if rank[x] > rank[y]:
    parent[y] = x
  else:
    parent[x] = y
    if rank[x] == rank[y]:
      rank[y] = rank[y] + 1

union(x, y):
  link(find(x), find(y))
```

`makeSet(x)` makes x its own parent with rank 0. `find(x)` compresses paths. `link(x, y)` joins two *roots*: the one of lower rank goes under the other, and on a tie `y`'s rank goes up. `union(x, y)` finds the two roots and links them. The `rank` array holds the ranks.

Why increment on a tie? When two trees of rank r merge, one root becomes the other's child, its deepest node drops a level, and the height can reach $r + 1$. With unequal ranks, the lower tree's deepened nodes stay within the taller tree's height.

Theorem 13.2 (Rank and number of elements)

Under union by rank, a root of rank r has at least 2^r elements in its tree. Hence no rank exceeds $\lfloor \log_2 n \rfloor$, and the height is $O(\log n)$ even without path compression.

Proof. By induction on the rank. A rank-0 root has at least $1 = 2^0$ element. A rank rises from r to $r + 1$ only when two roots of rank r merge, each tree having at least 2^r elements by hypothesis, so the result has at least 2^{r+1}. As there are only n elements, $2^r \le n$, so $r \le \log_2 n$, and the height, bounded by the rank, obeys the same bound. □

This bound is reached by merging 16 elements in pairs, fours, and eights: the final merge of two eight-element trees gives rank $4 = \log_2 16$.

With path compression, rank and height drift apart: compression flattens the tree without updating the rank, which remains only an *upper bound*. That is not a bug; rank is good enough for deciding links and costs nothing to maintain.

Common mistake

Forgetting the increment on equal ranks. Without `rank[ry]++`, all ranks stay 0, every merge takes the "equal" branch, and the rule degenerates into naive union: rank no longer tracks height, the $O(\log n)$ guarantee is gone, and a sequence like that of Section 13.5.1 builds a chain again.

13.5.4 Path compression

The second idea exploits `find` itself. `find(x)` walks from x to the root, and every node on that path has the same root; linking them all directly to it makes later `find`s take a single step.

Definition 13.7

Path compression: during `find(x)`, every node met on the path from x to the root gets the root as its parent.

The recursive `find` does this by itself, `parent[x] = find(parent[x])`: the recursion descends to the root, and on the way back each call makes that root its node's parent.

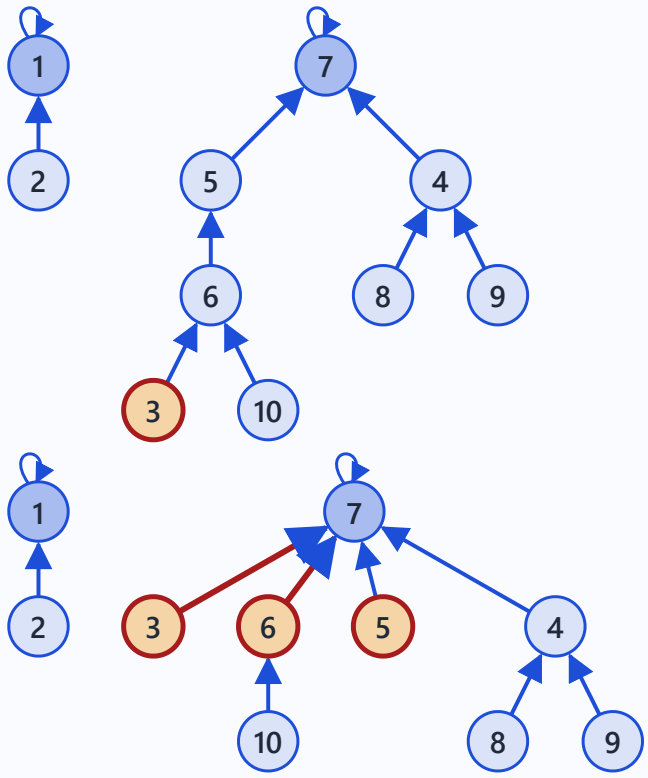


Figure 13.19: Path compression. Top, before: in the tree rooted at 7, the path of 3 is 3 → 6 → 5 → 7 (3 steps), and 10 is also linked to 6. Beside it is a separate tree rooted at 1 ({1, 2}). Bottom, after `find(3)` : 3 and 6 are linked directly to 7 (5 was already a child of 7); 10 is still linked to 6, but its path is shorter too.

Trace `find(3)` in Figure 13.19. Since `parent[3] = 6`, `find(6)` is called, then `find(5)`, then `find(7)`, which returns 7, the root. On the way back, `find(5)` writes `parent[5] = 7` (no change), `find(6)` writes `parent[6] = 7`, and `find(3)` writes `parent[3] = 7`: the path of 3 shrinks from three steps to one. 10, a child of 6, was off the path and keeps its parent, yet its path shrinks too, to 10 → 6 → 7.

Compression doesn't speed up the `find` that performs it, which does a little extra work (one write per node). The payoff comes later: a long path, once walked, is gone, and later queries on those nodes are cheap — a typical case for amortized analysis, which averages cost over a sequence of operations.

13.5.5 Complexity and the inverse Ackermann function

Implementation	`find`	`union`	Note
Naive (arbitrary linking)	O(n)	O(n)	the tree can become a chain
By size or by rank	O(log n)	O(log n)	height ≤ $\log_2$ n
Path compression only	O(log n) amortized		over *m* operations

Implementation	`find`	`union`	Note
Rank + path compression	**O(α(n)) amortized**		constant in practice

Table 13.18: Complexity of Union-Find implementations. Amortized bounds are averages per operation over sequences of m operations.

The first two rows follow from the previous sections, since a `union` costs two `find`s plus constant-time linking. The third says that path compression alone gives logarithmic average cost: one `find` may be slow, but long paths shorten once walked, bounding the total.

The last row is the chapter's most striking result. $\alpha(n)$ is the **inverse Ackermann function**. The Ackermann function $A_k(j)$ grows at a mind-boggling rate, each increase of k giving a function that applies the previous one to itself repeatedly. Its inverse $\alpha(n)$, the smallest k with $A_k(1) \geq n$, grows correspondingly slowly. With the CLRS definition, $A_4(1)$ vastly exceeds 10^{80}, an estimate of the number of atoms in the observable universe, so $\alpha(n) \leq 4$ for any conceivable input. In practice it is a constant.

Theorem 13.3 (Tarjan, 1975)

With union by rank and path compression, any sequence of m `makeSet`, `find`, and `union` operations on n elements runs in $O(m \cdot \alpha(n))$ time.

The proof is beyond these notes (CLRS gives it a section of its own), but three remarks help. The bound is *amortized*: one `find` can still take $O(\log n)$, but the total never exceeds $O(m \cdot \alpha(n))$. It is *tight*: Tarjan also showed sequences that force time proportional to $m \cdot \alpha$, so the algorithm never quite reaches linear time — hence "good but not linear" in his title. And memory is small: two integer arrays, `parent[n]` and `rank[n]` (or `size[n]`), $O(n)$.

Union-Find has many uses. Kruskal's minimum spanning tree algorithm uses `find` to ask whether an edge would close a cycle. A graph's connected components emerge from merging as edges arrive (Section 13.5.10). Image processing groups adjacent same-colored pixels into regions (region labeling), and networks ask whether two machines can reach each other.

13.5.6 Java: node-based Union-Find

Union-Find can also hold its elements as objects, each node storing its data, its rank, and a reference to its parent:

```
class SetNode {
  int data;
  int rank;   // bound on height
  SetNode parent;

  SetNode(int data) {
    this.data = data;
    this.parent = this;
    this.rank = 0;
  }
}

SetNode find(SetNode node) {
  if (node != node.parent)
    node.parent = find(node.parent); // compression
  return node.parent;
}
```

The constructor makes the node its own parent with rank 0, the object form of `makeSet`. `find` translates the pseudocode: a node that is not its own parent gets its parent's root as its new parent (path compression), and that root is returned. `node != node.parent` compares references, which is right here: the question is whether both point to the same object.

```
void union(SetNode x, SetNode y) {
  SetNode rootX = find(x);
  SetNode rootY = find(y);
  if (rootX == rootY) return;
  // union by rank
  if (rootX.rank < rootY.rank) {
    rootX.parent = rootY;
  } else if (rootX.rank > rootY.rank) {
    rootY.parent = rootX;
  } else { // equal: pick one
    rootX.parent = rootY;
    rootY.rank++;
  }
}
```

`union` finds both roots and returns if they are the same object. Otherwise the root of lower rank goes under the other; on a tie, `rootX` goes under `rootY`, whose rank goes up. Elements are used directly as objects, with no index map, at a price in memory and speed: every node is a separate object, and following references is slower than reading an array.

13.5.7 Java: array-based Union-Find

For elements *0..n − 1*, two integer arrays suffice. This is the form most used in practice:

```
class UnionFind {
  private final int[] parent, rank;
  UnionFind(int n) {
    parent = new int[n];  rank = new int[n];
    for (int i = 0; i < n; i++) parent[i] = i;
  }
  int find(int x) {
    if (parent[x] != x) parent[x] = find(parent[x]);
    return parent[x];
  }
  boolean union(int x, int y) {
    int rx = find(x), ry = find(y);
    if (rx == ry) return false;
    if (rank[rx] < rank[ry]) parent[rx] = ry;
    else if (rank[rx] > rank[ry]) parent[ry] = rx;
    else { parent[rx] = ry; rank[ry]++; }
    return true;
  }
}
```

Every element starts as its own parent, and `rank` starts at zero, Java's default. `find` compresses paths in one line. `union` returns `false` if the two roots coincide; otherwise it links them by rank and returns `true`.

The `boolean` result is deliberate: it says whether the call actually merged two different sets. Kruskal's algorithm asks exactly this before adding an edge — an edge whose endpoints already share a set would close a cycle and is skipped — and component counting (Section 13.5.10) relies on it too.

For example, after `UnionFind uf = new UnionFind(5); uf.union(0, 1); uf.union(3, 4);`, `uf.find(1) == uf.find(0)` is `true` and `uf.find(1) == uf.find(3)` is `false`. In the first merge the ranks are equal, so 0 goes under 1 and `rank[1] = 1`; in the second, 3 goes under 4.

> **Common mistake**
>
> Ignoring the recursion depth of `find` for very large *n*. With union by rank the height is *O(log n)*, which is harmless. But with path compression alone, or naive linking, the height can approach *n*, and on a chain of hundreds of thousands of elements a recursive `find` throws `StackOverflowError`. Write `find` with a loop instead (Section 13.5.9).

13.5.8 Code trace: find and union

We trace the array-based structure on 6 elements with `union(0, 1)`, `union(2, 3)`, `union(0, 2)`, `union(4, 0)`, `union(1, 4)`, using these methods:

```
int find(int x) {
  if (parent[x] != x) parent[x] = find(parent[x]);
  return parent[x];
}
boolean union(int x, int y) {
  int rx = find(x), ry = find(y);
  if (rx == ry) return false;
  if (rank[rx] < rank[ry]) parent[rx] = ry;
  else if (rank[rx] > rank[ry]) parent[ry] = rx;
  else { parent[rx] = ry; rank[ry]++; }
  return true;
}
```

Call	`rx`, `ry`	Ranks	Decision	Returns	`parent`	`rank`
start	—	—	—	—	[0, 1, 2, 3, 4, 5]	[0, 0, 0, 0, 0, 0]
`union(0, 1)`	0, 1	0 = 0	0 → 1, `rank[1]++`	`true`	[1, 1, 2, 3, 4, 5]	[0, 1, 0, 0, 0, 0]
`union(2, 3)`	2, 3	0 = 0	2 → 3, `rank[3]++`	`true`	[1, 1, 3, 3, 4, 5]	[0, 1, 0, 1, 0, 0]
`union(0, 2)`	1, 3	1 = 1	1 → 3, `rank[3]++`	`true`	[1, 3, 3, 3, 4, 5]	[0, 1, 0, 2, 0, 0]
`union(4, 0)`	4, 3	0 < 2	4 → 3 (path compression: 0 → 3)	`true`	[3, 3, 3, 3, 3, 5]	[0, 1, 0, 2, 0, 0]
`union(1, 4)`	3, 3	—	same set	`false`	[3, 3, 3, 3, 3, 5]	[0, 1, 0, 2, 0, 0]

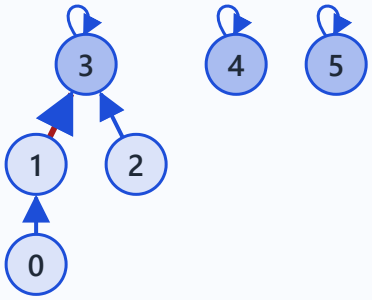
Table 13.19: Trace of five `union` calls on `new UnionFind(6)`. The arrays show the values after each call.

In the first two calls all elements are roots, so each `find` returns at once. The ranks are equal, so the `else` branch runs (`parent[rx] = ry`, `rank[ry]++`): {0, 1} gets root 1 and {2, 3} root 3, both of rank 1.

In `union(0, 2)`, `find(0)` sees `parent[0] = 1 ≠ 0`, calls `find(1)`, which returns the root 1, and rewrites `parent[0] = 1`, unchanged. Likewise `find(2) = 3`. The roots 1 and 3 have equal rank, so 1 goes under 3 and `rank[3] = 2`. The tree now has root 3 with children 2 and 1, and 0 under 1: height 2, as rank 2 says.



i	0	1	2	3	4	5
parent	1	1	2	3	4	5
rank	0	1	0	0	0	0

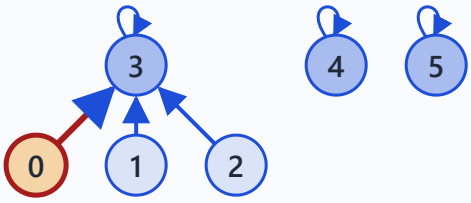


i	0	1	2	3	4	5
parent	1	3	3	3	4	5
rank	0	1	0	2	0	0

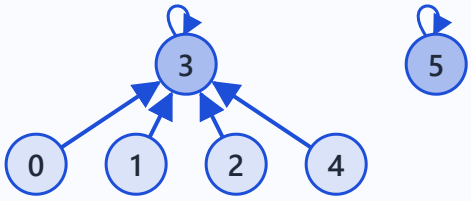
Figure 13.20: Two moments from the first half of the trace. (a) After `union(0, 1)` : 0 is under 1; `rank[1] = 1` . (b) After `union(0, 2)` : two roots of rank 1 have merged, 1 → 3; `rank[3] = 2` . The path of 0 is 0 → 1 → 3.

`union(4, 0)` shows path compression at work. `find(4)` returns 4; `find(0)` recurses 0 → 1 → 3, and on the way back writes `parent[1] = 3` (unchanged) and `parent[0] = 3` , moving 0 directly under the root. Since `rank[4] = 0 < rank[3] = 2` , 4 is attached to 3, and neither rank nor height changes.

In `union(1, 4)` , both `find` s return 3, so `rx == ry` and the method returns `false` — for Kruskal's algorithm, an edge that would close a cycle.



i	0	1	2	3	4	5
parent	3	3	3	3	4	5
rank	0	1	0	2	0	0



i	0	1	2	3	4	5
parent	3	3	3	3	3	5
rank	0	1	0	2	0	0

Figure 13.21: Two moments from the second half of the trace. (a) After `find(0)` inside `union(4, 0)` : path compression has linked 0 directly to the root. (b) The result: {0, 1, 2, 3, 4} is one set with root 3, and {5} is a separate set. `rank[3]` is still 2, but the true height is now 1.

The trace ends with rank and height apart: `rank[3] = 2`, but with 0 moved up the true height is 1, and the rank is still a valid upper bound. Together, rank and path compression cost about $O(\alpha(n))$ per operation.

13.5.9 Iterative find with path compression

To avoid recursion, `find` can use two loops: the first finds the root, the second walks the path again and links each node to the root.

```
int find(int x) {
  int root = x;
  while (parent[root] != root) root = parent[root];
  while (parent[x] != root) {    // path compression
    int next = parent[x];
    parent[x] = root;
    x = next;
  }
  return root;
}
```

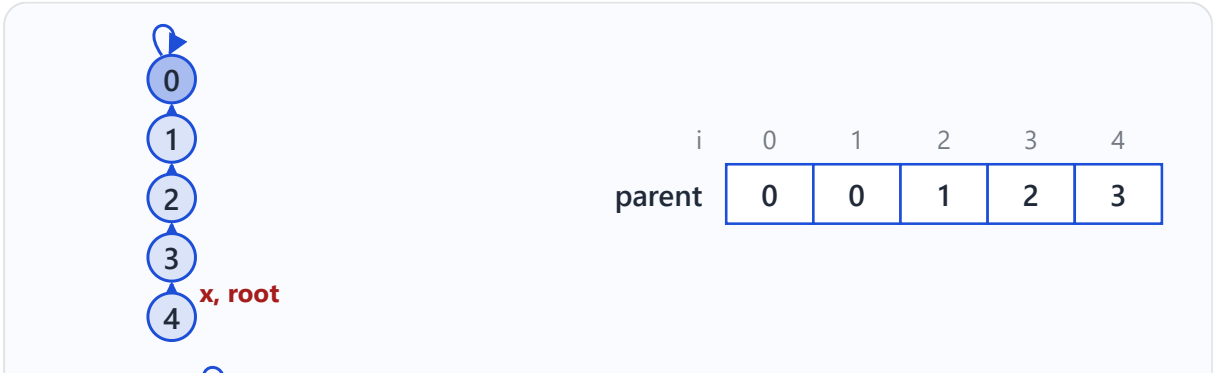
The first loop moves `root` up from x until it reaches the node that points to itself. The second restarts at `x`; each step saves the old parent in `next`, writes `parent[x] = root`, and moves up with `x = next`. The old parent must be saved first, or the rest of the path is lost. The loop stops when `parent[x]` is already the root.

Here is `find(4)` on the chain `parent = [0, 0, 1, 2, 3]` ($4 \to 3 \to 2 \to 1 \to 0$), left by bad merges.

Loop	Step	x	root	next	parent
—	`root = x`	4	4	—	[0, 0, 1, 2, 3]
1st	`parent[4] = 3 ≠ 4`	4	3	—	[0, 0, 1, 2, 3]
1st	`parent[3] = 2 ≠ 3`	4	2	—	[0, 0, 1, 2, 3]
1st	`parent[2] = 1 ≠ 2`	4	1	—	[0, 0, 1, 2, 3]
1st	`parent[1] = 0 ≠ 1`; then `parent[0] = 0`: stop	4	0	—	[0, 0, 1, 2, 3]
2nd	`parent[4] = 3 ≠ 0` → compress	3	0	3	[0, 0, 1, 2, **0**]
2nd	`parent[3] = 2 ≠ 0` → compress	2	0	2	[0, 0, 1, **0**, 0]
2nd	`parent[2] = 1 ≠ 0` → compress	1	0	1	[0, 0, **0**, 0, 0]
2nd	`parent[1] = 0 = root`: stop	1	0	—	[0, 0, 0, 0, 0]

Table 13.20: Trace of iterative `find(4)`, `parent = [0, 0, 1, 2, 3]`.

The first loop reaches the root in four steps; the second relinks 4, 3, and 2 to 0 and stops at 1, already a child of the root. The method returns 0, and every later `find` on these nodes takes one step. Without recursion, no chain is long enough to overflow the stack.



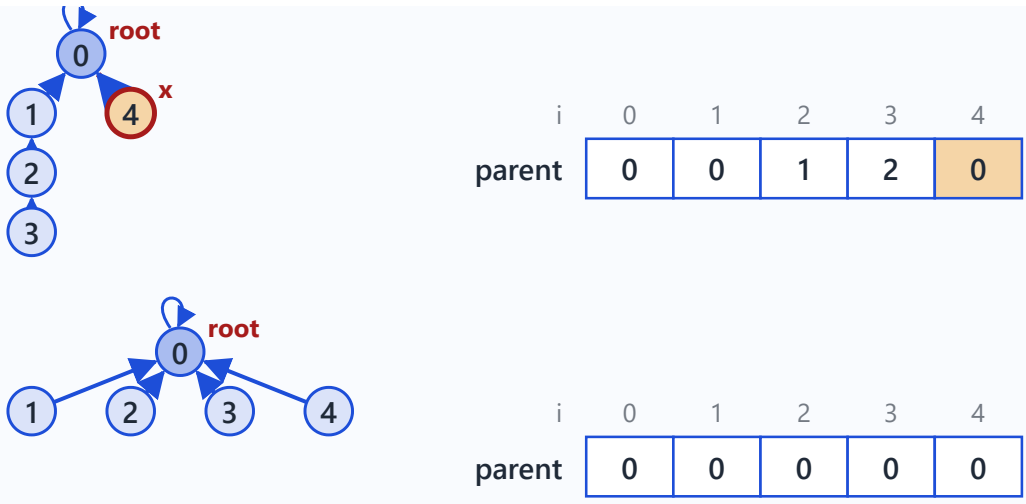


Figure 13.22: Iterative `find(4)`. Top, the start: the chain 4 → 3 → 2 → 1 → 0. Middle, the first step of the second loop: `next = 3` has been saved, and 4 is linked directly to 0. Bottom, the result: every node is linked to the root, `parent = [0, 0, 0, 0, 0]`.

Cost. Each loop walks the path once, so the time is still *O(path length)*. The compression matches the recursive version, but extra memory is *O(1)* rather than one call frame per step.

13.5.10 Application: counting connected components

A **connected component** of a graph is a maximal group of nodes that can reach one another by paths. How many components does a graph with n nodes and a given edge list have? Union-Find answers naturally: each node starts alone, and each edge merges its endpoints' components.

```
int countComponents(int n, int[][] edges) {
  UnionFind uf = new UnionFind(n);
  int count = n;        // n singleton sets
  for (int[] e : edges)
    if (uf.union(e[0], e[1]))
      count--;          // two components merged
  return count;
}
```

`count` starts at n, every node being its own component. A `union` that returns `true` has merged two components, so the count drops by one; `false` means the endpoints already share a component, the edge closes a cycle, and the count stays. Again, `union`'s return value is the heart of the method.

A trace with 7 nodes and the edges `(0, 1)`, `(1, 2)`, `(3, 4)`, `(0, 2)`, `(5, 6)`:

Edge	Roots	`union`	`count`	`parent` (after)	Components
(start)	—	—	7	[0, 1, 2, 3, 4, 5, 6]	seven singletons
(0, 1)	0, 1	`true`	6	[1, 1, 2, 3, 4, 5, 6]	{0, 1} and five singletons
(1, 2)	1, 2	`true`	5	[1, 1, 1, 3, 4, 5, 6]	{0, 1, 2} and four singletons
(3, 4)	3, 4	`true`	4	[1, 1, 1, 4, 4, 5, 6]	{0, 1, 2}, {3, 4}, {5}, {6}
(0, 2)	1, 1	`false`	4	unchanged	unchanged (cycle)
(5, 6)	5, 6	`true`	3	[1, 1, 1, 4, 4, 6, 6]	{0, 1, 2}, {3, 4}, {5, 6}

Table 13.21: Trace of `countComponents(7, edges)`.

At `(1, 2)`, 1 has rank 1 and 2 rank 0, so 2 goes under 1. At `(0, 2)`, both endpoints have root 1, so `union` returns `false` and `count` stays 4: this edge closes the triangle 0–1–2. Three components remain, and the method returns 3.

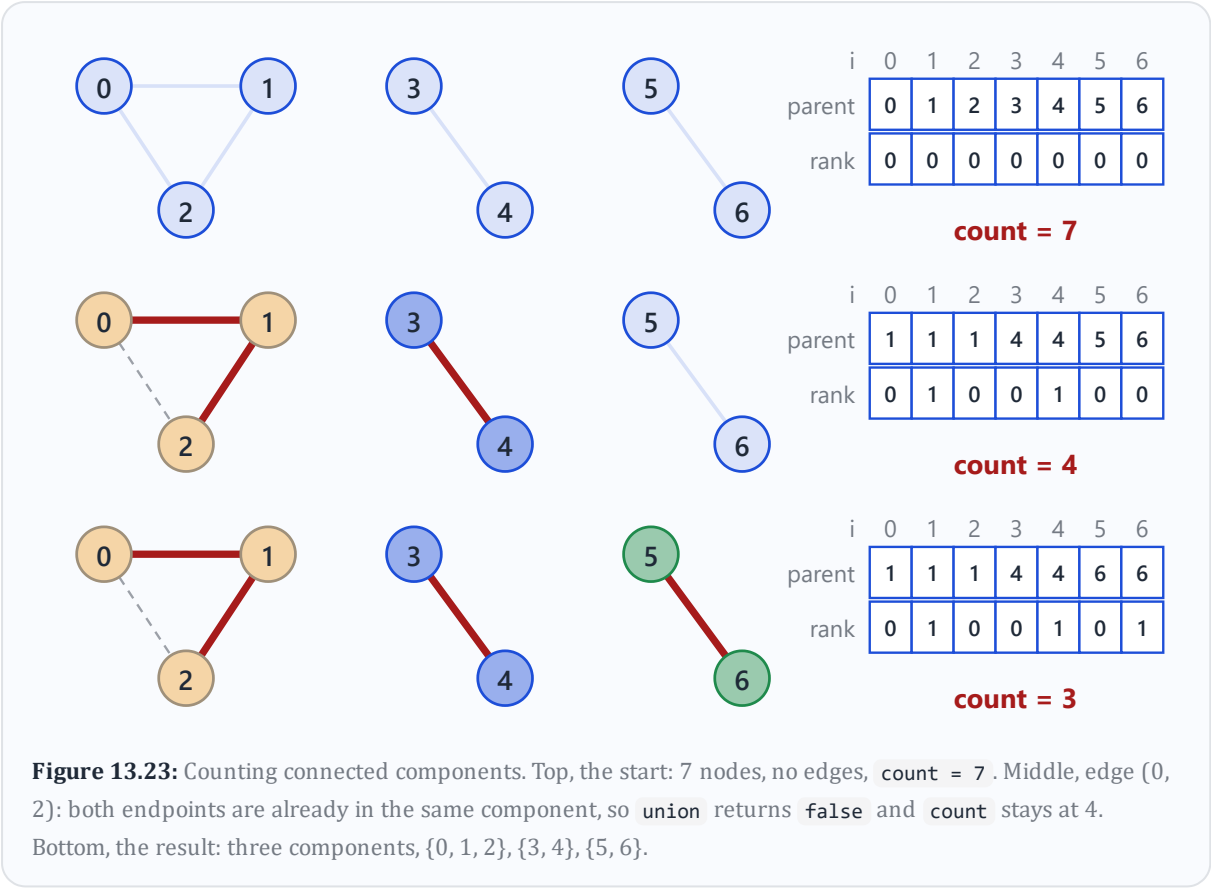


Figure 13.23: Counting connected components. Top, the start: 7 nodes, no edges, `count = 7`. Middle, edge (0, 2): both endpoints are already in the same component, so `union` returns `false` and `count` stays at 4. Bottom, the result: three components, {0, 1, 2}, {3, 4}, {5, 6}.

Cost. Building takes $O(n)$, and each of the m edges one `union` (two `find`s): $O(n + m \cdot \alpha(n))$ in all. Depth-first search answers in $O(n + m)$, but Union-Find shines at **dynamic connectivity**: as edges arrive one at a time, the count is current after each, while depth-first search would start over. The graph is connected exactly when `count == 1`.

Example 13.3

In a 6-node graph, the edges `(0, 1)`, `(1, 2)`, `(3, 4)`, `(2, 0)` arrive in order. Which `union` returns `false`, and how many components remain?

1. **Start.** `count = 6`.
2. **(0, 1):** roots 0 and 1, different. They merge; `count = 5`.
3. **(1, 2):** 1's root is 1, 2's is 2. They merge; `count = 4`.
4. **(3, 4):** different roots; `count = 3`.
5. **(2, 0):** `find(2) = 1`, `find(0) = 1`: same root, so `union` returns `false`; the edge closes the cycle 0–1–2.
6. **Result.** Three components: {0, 1, 2}, {3, 4}, and {5}, which has no edges.

13.6 Common mistakes

The chapter's mistakes fall into two groups: the first three concern Java sets, the last three Union-Find and sorted sets.

Mistake	What happens	Fix
Modifying an element inside a set	An object whose hashed field changes stays in its old bucket; `contains` looks in the new bucket and returns `false`.	Don't change the hashed fields of objects in a set; if you must, remove the object, change it, and add it again.
`equals` **without** `hashCode`	Two "equal" objects land in different buckets and both end up in the `HashSet`.	Override both together, so equal objects produce equal hash codes.
Removing while iterating	`set.remove(x)` inside for-each → `ConcurrentModificationException`.	`set.removeIf(x -> x % 2 == 0)` or the iterator's `remove()` method.
Linking items instead of roots	`parent[x] = y`: if x is not a root, its tree splits.	`parent[find(x)] = find(y)`
Forgetting the increment on equal ranks	Rank stops tracking height; the *O(log n)* guarantee is lost.	`rank[ry]++` on equal ranks.
`compare` **inconsistent with** `equals` **in a** `TreeSet`	Of distinct objects with `compare(a, b) == 0`, only one is kept.	Make the ordering consistent with `equals` (`thenComparing`).

Table 13.22: Common mistakes when working with sets and Union-Find.

The first is insidious because nothing fails; answers are just wrong. Add a `Point` whose hash code comes from `x` and `y` while it is *(1, 2)*, then set `p.x = 5`. Its hash code changes, but it stays in the old bucket, so `set.contains(p)` looks elsewhere and fails, though the set's size is still 1. That is why immutable objects (`String`, `Integer`, `record`s) are preferred as set elements and map keys.

The third arises because a for-each loop runs on an iterator that tracks structural changes; if the set changes by another route mid-loop, the iterator throws at its next step. `removeIf` is safe because it removes through the iterator itself.

13.7 Summary

- A **set** is an unordered collection of distinct elements, with insertion, removal, and membership as basic operations. Union ($\cup$), intersection ($\cap$), difference ($-$, not commutative), and subset ($\subseteq$) are the set operations.
- Java's `Set` interface: `add` returns `false` for an element already present, detecting repeats in one call; `addAll`, `retainAll`, `removeAll` work **in place**, so copy a set first to keep it.
- A **hash-based set** puts each element in the bucket given by its hash code, chaining collisions. With a bounded load factor, `add`, `contains`, `remove` are $O(1)$ on average, $O(n)$ with a bad hash function. Override `equals` and `hashCode` together.
- A **tree-based set** (`TreeSet`, a red-black tree) does everything in $O(\log n)$, including sorted queries such as `first`, `floor`, `ceiling`, `subSet`; equality is decided by `compare`.
- **Sorted arrays** give union and intersection in one two-pointer pass, $O(n + m)$; on a common element both pointers advance.
- A **bit vector** stores a set over a small universe in U bits; single-element operations, and set operations for $U \le 32$, are single instructions, and `BitSet` operations take $O(U / 64)$. Sparse sets use less memory as hash sets.
- A **multiset** records multiplicities in an element → counter map, deleting zero counters; `size()` counts repeats. A **multimap** maps a key to a list of values (`Map<K, List<V>>`).
- **Disjoint sets** partition elements into groups. **Union-Find** keeps each group as a tree whose root represents it; `find` climbs to the root, `union` links two roots. Objects are first mapped to indices.
- **Quick-Find** has $O(1)$ `find` but array-scanning `union`s: $n - 1$ merges cost $O(n^2)$. A naive parent tree can become a chain, with $O(n)$ `find`.
- **Union by size or by rank** bounds the height by $\lfloor \log_2 n \rfloor$ (a rank-r tree has at least 2^r elements). **Path compression** links every node on `find`'s path to the root. Together they run m operations in $O(m \cdot \alpha(n))$ (Tarjan, 1975), with $\alpha(n) \le 4$.
- `union`'s `boolean` says whether it really merged: each `true` decrements the component count, and in Kruskal's algorithm a `false` marks a cycle-forming edge.

Structure	Basic operations	Cost	When?
`HashSet`	`add`, `remove`, `contains`	O(1)*	order not needed
`TreeSet`	+ `first`, `floor`, range	O(log n)	sorted access needed
Bit vector	bit operations	O(1) / O(U / 64)	small, dense universe
Multiset	`add`, `count`, `remove`	O(1)*	repeat counts matter

Structure	Basic operations	Cost	When?
Union-Find	`find`, `union`	O(α(n))	disjoint sets, connectivity queries

Table 13.23: Summary of the structures in this chapter. * average case; Union-Find costs are amortized, with rank + path compression.

13.8 Key terms

Each term links to the course glossary, which gives its Turkish equivalent and its use in other chapters.

set · abstract data type (ADT) · union · intersection · difference · subset · hash table · hash code · bucket · collision · separate chaining · load factor · binary search tree (BST) · red-black tree · inorder · merge sort · bit vector · multiset · multimap · map · disjoint sets (union-find) · parent · root · height · union by size · rank · union by rank · path compression · amortized analysis · connected component · Kruskal's algorithm · iterator

13.9 Test Yourself

1. For $A = \{1, 3, 5, 7, 9\}$ and $B = \{3, 6, 9\}$, what are $A \cup B$, $A \cap B$, $A - B$, and $B - A$? Are $\{3, 9\} \subseteq B$ and $B \subseteq A$ true?

   Answer

   $A \cup B = \{1, 3, 5, 6, 7, 9\}$, $A \cap B = \{3, 9\}$, $A - B = \{1, 5, 7\}$, $B - A = \{6\}$. $\{3, 9\} \subseteq B$ is true; $B \subseteq A$ is false, because $6 \notin A$. Check: $|A \cup B| = 5 + 3 - 2 = 6$.

2. What does `firstRepeat(new int[]{5, 1, 9, 1, 5})` return? How many times is `add` called before the method returns?

   Answer

   1. The first three `add`s (5, 1, 9) return `true`, leaving {1, 5, 9}; the fourth, `add(1)`, returns `false`, so the method returns 1 after 4 `add`s. The final 5 is never examined: it repeats too, but later.

3. The values 14, 3, 21, 10, 3 are added in order to a chained hash set with 7 buckets (bucket = `x % 7`). What does each `add` return, and what is the final state of the buckets? How many

comparisons does a following `contains(17)` make?

Answer

Buckets: 14 → 0, 3 → 3, 21 → 0, 10 → 3, 3 → 3, so the calls return `true, true, true, true, false`. Bucket 0 holds 14 → 21, bucket 3 holds 3 → 10, the rest are empty, and `n = 4`. `contains(17)` goes to bucket *17 mod 7 = 3*, compares with 3 and 10 (2 comparisons), and returns `false`.

4. `unionIntersection` is called with `a = [2, 5, 8, 11]` and `b = [1, 5, 11, 12, 15]`. What are `uni` and `both`? How many times does the main loop run?

Answer

Passes: 2 > 1 → add 1; 2 < 5 → add 2; 5 = 5 → common; 8 < 11 → add 8; 11 = 11 → common. Then `i = 4 = a.length`, ending the main loop after 5 passes, and the leftover 12 and 15 join the union: `uni = [1, 2, 5, 8, 11, 12, 15]`, `both = [5, 11]`.

5. The set `a` is built from the bits {0, 4, 6} and `b` from {4, 5}. What are the numeric values of `a`, `b`, `a | b`, `a & b`, and `a & ~b`, and which sets do they represent?

Answer

a = 1 + 16 + 64 = 81 (0101 0001), *b = 16 + 32 = 48* (0011 0000). `a | b = 113` (0111 0001) → {0, 4, 5, 6}. `a & b = 16` → {4}. `a & ~b = 65` (0100 0001) → {0, 6}. Four bits are set in the union: `Integer.bitCount(a | b) = 4`.

6. The letters of the word "tattoos" are added to a `Multiset<Character>`. Then `remove('t')` is called twice and `remove('a')` once. What do `count('t')`, `count('a')`, and `size()` return? How many entries are left in the map?

Answer

After the insertions the map is {t:3, a:1, o:2, s:1}, `n = 7`. The two `remove('t')` calls take the counter 3 → 2 → 1; `remove('a')` deletes its entry. `count('t') = 1`, `count('a') = 0`, `size() = 4`, and three entries remain: {t:1, o:2, s:1}.

7. For the `TreeSet` *S = {4, 9, 13, 18, 22, 30}*, what do `floor(20)`, `ceiling(20)`, `subSet(9, 22)`, and `floor(3)` return?

Answer

`floor(20) = 18` (largest ≤ 20), `ceiling(20) = 22` (smallest ≥ 20), `subSet(9, 22) = [9, 13, 18]` (22 excluded). `floor(3)` returns `null`, since no element is ≤ 3; assigning it to an `int` throws a `NullPointerException`.

8. On an array-based `UnionFind(6)`, the calls `union(0, 1)`, `union(1, 2)`, `union(3, 4)`, `union(4, 5)`, `union(2, 5)`, `union(0, 3)` are made in order. What does the last call return? What are the final `parent` and `rank` arrays?

   Answer

   `(0, 1)`: equal ranks, 0 → 1, `rank[1] = 1`. `(1, 2)`: 2 → 1. `(3, 4)`: 3 → 4, `rank[4] = 1`. `(4, 5)`: 5 → 4. `(2, 5)`: roots 1 and 4 of equal rank; 1 → 4, `rank[4] = 2`. `(0, 3)`: `find(0)` walks 0 → 1 → 4 and compresses 0 onto 4; `find(3) = 4`; same root, so `false`. Final: `parent = [4, 4, 1, 4, 4, 4]`, `rank = [0, 1, 0, 0, 2, 0]`. 2 still points to 1, since no `find` passed through it.

9. With `parent = [0, 0, 1, 2, 2, 4]`, the iterative `find(5)` is called. In how many steps is the root found, and what is the final array? Why doesn't 3's parent change?

   Answer

   The path 5 → 4 → 2 → 1 → 0 finds the root in 4 steps. The second loop links 5, 4, and 2 to 0 (1 already is), giving `[0, 0, 0, 2, 0, 0]`. 3, a child of 2, is not on the path walked, and compression changes only that path; still, 3's path shrinks to 3 → 2 → 0.

10. Under union by size (with the second root going under the first on ties), what is the greatest possible height of a tree of 8 elements? Give a sequence of `union`s that produces it.

    Answer

    At most $\lfloor \log_2 8 \rfloor = 3$, by the theorem. `union(0,1)`, `union(2,3)`, `union(4,5)`, `union(6,7)` make four pairs; `union(0,2)` and `union(4,6)` two fours; `union(0,4)` one tree, with `parent = [0, 0, 0, 2, 0, 4, 4, 6]` and the path 7 → 6 → 4 → 0 of 3 steps. Each merge joined equal-sized trees, raising the height by one.

13.10 Further reading

- M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, *Data Structures and Algorithms in Java* (6th ed., 2014), Sections 10.5 and 14.7: the set, multiset, and multimap ADTs, the generic merge, and Union-Find within Kruskal's algorithm, in Java close to that of these notes.
- R. Sedgewick and K. Wayne, *Algorithms* (4th ed., 2011), Section 1.5: the path from Quick-Find through Quick-Union and weighted union to path compression, compared with experimental measurements.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms* (4th ed., 2022), Chapter 19: disjoint-set forests, union by rank, path compression, and a formal proof of the $O(m \cdot \alpha(n))$ bound.

- R. E. Tarjan, "Efficiency of a Good But Not Linear Set Union Algorithm" (1975): the original paper showing that rank and path compression together give the inverse Ackermann bound, and that the bound is tight.

13.11 References

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022.
2. B. A. Galler, M. J. Fischer, "An improved equivalence algorithm," *Communications of the ACM*, 7(5), 301–303, 1964.
3. M. T. Goodrich, R. Tamassia, M. H. Goldwasser, *Data Structures and Algorithms in Java*, 6th ed., Wiley, 2014.
4. Oracle, *Java SE 17 API Specification*: documentation for `java.util.Set`, `HashSet`, `TreeSet`, and `BitSet`.
5. R. Sedgewick, K. Wayne, *Algorithms*, 4th ed., Addison-Wesley, 2011.
6. R. E. Tarjan, "Efficiency of a Good But Not Linear Set Union Algorithm," *Journal of the ACM*, 22(2), 215–225, 1975.

PART IV · CHAPTER 14

Graphs

Cities and the roads between them, people in a social network and their friendships, the courses in a department and their prerequisites: all three are described by the same abstract structure, a set of objects together with pairwise relationships among them. That structure is a graph. A tree gives every node one parent and forbids cycles; a graph drops both restrictions, so a vertex may be reachable along many routes, and a route may come back to where it started. We begin with the basic vocabulary of graphs, then look at how a graph is stored in memory and how that choice shapes the cost of each operation. Next come breadth-first and depth-first search, together with the algorithms built on them: finding components, testing bipartiteness, topological sorting, and cycle detection. The last two topics are shortest paths (BFS for unweighted graphs, Dijkstra for weighted ones) and minimum spanning trees (Kruskal and Prim).

Code for this chapter: `github.com/sercankulcu/data-structures-java/Bolum14/src`

LEARNING OUTCOMES

Basic concepts
Explain vertices, edges, direction, weight, degree, paths, cycles, and connectivity; use the handshake lemma to count edges and degrees.

Representations
Store a graph as an edge list, an adjacency list, or an adjacency matrix; compare their memory and time costs and pick the one that suits a problem.

Traversal and ordering
Carry out BFS and DFS by hand and in code; find connected components, test bipartiteness, topologically sort a directed acyclic graph (DAG), and detect directed cycles.

Shortest paths and MSTs
Apply BFS to unweighted shortest paths, Dijkstra's algorithm to weighted ones, and Kruskal's and Prim's algorithms to minimum spanning trees, and justify which one a problem calls for.

14.1 Basic concepts

Graph algorithms speak a small vocabulary: vertex, edge, degree, path, cycle, connectivity. The words mean roughly what they do in everyday speech, but each has a precise definition, and correctness arguments rest on it.

14.1.1 What is a graph?

A city map shows intersections, the roads between them, and much detail besides: lengths, slopes, lanes. To answer "which intersection can I reach from which?" two facts suffice: the intersections, and which pairs are joined directly by a road. The abstract structure built from these two facts is a **graph**.

Definition 14.1

A **graph** is a pair $G = (V, E)$, where V is a finite set of **vertices** (or **nodes**) and E is a set of **edges**. Each edge joins two vertices and is written (u, v). We write $n = |V|$ for the number of vertices and $m = |E|$ for the number of edges.

$|V|$ is the size of V. Throughout, n counts vertices, m counts edges, and costs are stated in these terms: "$O(n + m)$ time" grows with vertices plus edges.

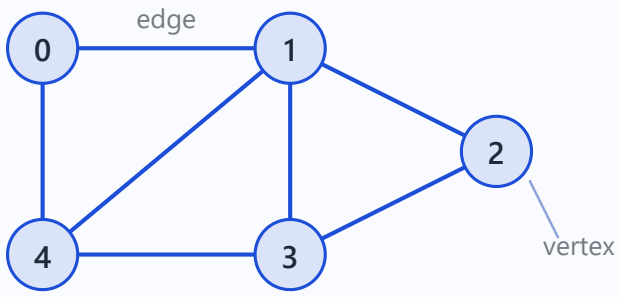


Figure 14.1: An undirected graph with five vertices, numbered 0 to 4, and seven edges. Circles are vertices; lines are edges.

Example 14.1

In set notation, the graph of Figure 14.1 is:

- $V = \{0, 1, 2, 3, 4\}$, so $n = 5$.
- $E = \{(0,1), (0,4), (1,2), (1,3), (1,4), (2,3), (3,4)\}$, so $m = 7$.

The edges have no direction: *(0,1)* and *(1,0)* are one edge, listed once. Picture and sets carry the same information; vertex positions and line shapes are not part of the graph.

A graph generalizes a tree, which has a root, one parent per non-root node, and no cycles. A graph requires none of this:

- **No root.** No vertex is special; an algorithm must be told where to start.
- **Cycles.** Edges can lead back to the start; in Figure 14.1, 1 → 2 → 3 → 1 is one.
- **Several routes.** From 0 to 3 you can go 0 → 1 → 3 or 0 → 4 → 3.

A tree models one-to-many relationships; a graph models many-to-many ones. A person has many friends, each with many friends; a course has several prerequisites and is a prerequisite of several others. The algorithms pay for this flexibility: they may reach a vertex twice and must notice.

Common mistake

Mistaking a drawing of a graph for the graph. The same graph can be drawn in many ways; moving vertices or letting two edges cross creates no new vertex or edge, and a crossing point is not a vertex. A graph is defined by its sets V and E alone.

14.1.2 Vertices and edges

A **vertex** stands for a modeled object: a city, a person, a computer, a web page, a course. It may carry a label, name, or position, but algorithms prefer vertices numbered 0 to $n - 1$, so per-vertex data can live in arrays indexed by vertex.

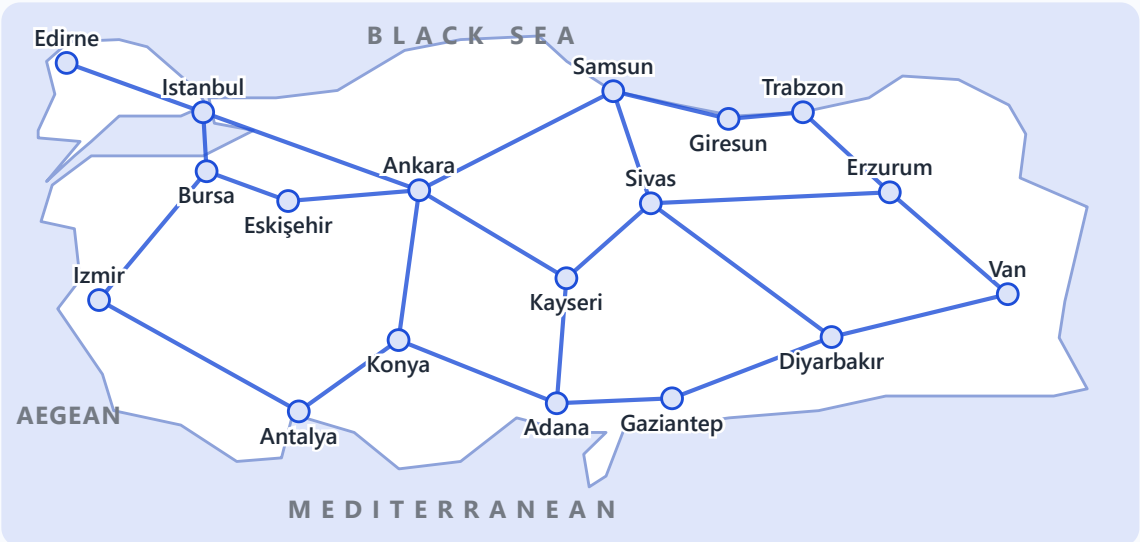


Figure 14.2: A simplified drawing of the Turkish highway network. Cities are vertices; the main roads between them are edges.

Modeled as a graph, “what is the shortest route from Istanbul to Van?” becomes a **graph algorithm** problem; with road lengths on the edges, Dijkstra's algorithm solves it (14.4.4). Modeling keeps the essence and drops the rest: paving is irrelevant to the shortest route, length is not.

An **edge** stands for a relationship: a road, a network link, a friendship. If *(v, w)* is an edge, *v* and *w* are **adjacent**, each a **neighbor** of the other, and the edge is **incident** to both. In Figure 14.1, 1 and 3 are neighbors; 0 and 2 are not.

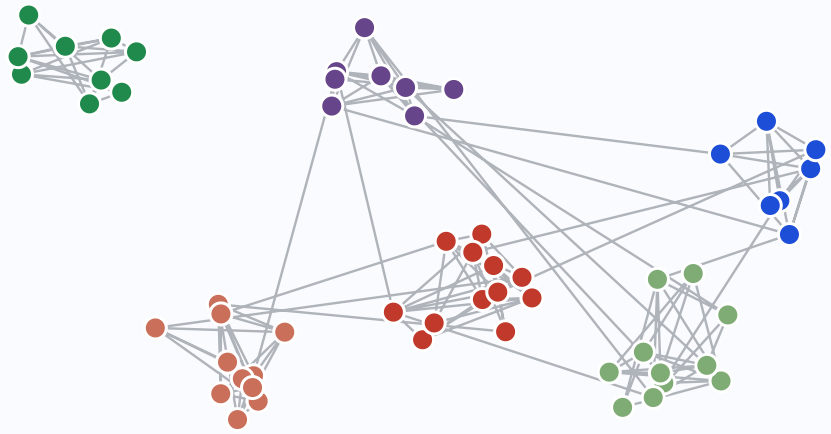


Figure 14.3: A social network. People are vertices and friendships are edges. Colors mark tightly knit communities, with few edges between them.

Figure 14.3 shows a typical trait of real networks: many vertices, few neighbors each. Millions of members, but at most a few thousand friends per person, so the edges are a tiny fraction of those possible. Such graphs are **sparse**, and sparsity decides how to store them (14.2.6).

14.1.3 Degree and the handshake lemma

Definition 14.2

In an undirected graph, the **degree** of a vertex *v*, written *deg(v)*, is the number of edges incident to *v*. A vertex of degree 0 is an **isolated vertex**. An edge joining a vertex to itself is a **self-loop**, and it contributes 2 to the degree.

An intersection's degree is the number of roads meeting there. A road that leaves an intersection and returns to it is attached at both ends, which is why a self-loop counts twice.

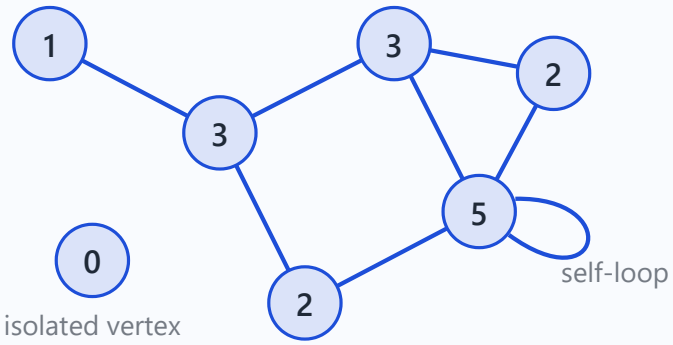


Figure 14.4: Each vertex is labeled with its degree: 1, 3, 3, 2, 5, 2, and 0. The degree-5 vertex at the lower right has a self-loop; the degree-0 vertex on the left is isolated.

The degrees in Figure 14.4 sum to $1 + 3 + 3 + 2 + 5 + 2 + 0 = 16$, and the graph has seven ordinary edges plus a self-loop, eight in all. The sum being exactly twice the edge count is no accident.

Theorem 14.1 (Handshake lemma)

In every undirected graph, the degrees sum to twice the number of edges: $\Sigma\, deg(v) = 2m$.

Proof. Ask how often each edge is counted. An edge (u, v) adds 1 to $deg(u)$ and 1 to $deg(v)$ and touches no other degree; a self-loop has $u = v$ and by definition also adds 2. Every edge adds exactly 2, so m edges add $2m$. □

At a party, the numbers of hands each guest shook add up to twice the number of handshakes, since each handshake involves two hands. Two consequences:

- The degrees give the edge count: $m = \Sigma\, deg(v) / 2$. Degrees summing to 30 mean 15 edges.
- The sum is even, so the number of odd-degree vertices is even. No graph has degrees 3, 3, 2, 2, 1: their sum is 11.

Common mistake

Forgetting to divide the degree sum by two. Each edge is counted once at each end, so the sum is twice the number of edges. The same slip appears in memory estimates: the adjacency lists of an undirected graph hold $2m$ entries, not m.

14.1.4 Directed graphs

Friendship is symmetric, and a two-way road runs both ways. Other relationships go one way: a web page may link to one that does not link back, a prerequisite does not work in reverse, a one-way street runs one way. These are modeled by a **directed graph**, or **digraph**.

Definition 14.3

In a **directed graph**, each edge is an ordered pair *(u, v)*, drawn as an arrow from *u* to *v*. *u* is the edge's **tail** (its origin) and *v* its **head** (its destination); *(u, v) ≠ (v, u)*. The **in-degree** of a vertex is the number of arrows entering it, and its **out-degree** the number leaving it.

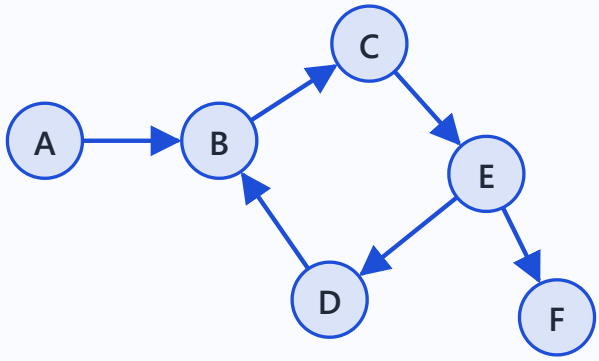


Figure 14.5: A directed graph with six vertices and the arrows A → B, B → C, C → E, E → D, D → B, and E → F.

Vertex	A	B	C	D	E	F	Total
in-degree	0	2	1	1	1	1	6
out-degree	1	1	1	1	2	0	6

Table 14.1: In- and out-degrees of the graph in Figure 14.5.

Count the arrows: two enter B (from A and D), so its in-degree is 2; two leave E (to D and F), so its out-degree is 2. Nothing enters A or leaves F.

Both rows sum to 6, the number of edges: each arrow adds 1 to its tail's out-degree and 1 to its head's in-degree, so Σ *in-degree* = Σ *out-degree* = m. In directed graphs, the undirected $2m$ splits into two sums.

Reachability depends on direction too. In Figure 14.5 A reaches B, but nothing reaches A. From B you can go to C, E, D, and back to B, so B, C, E, and D reach one another, while A and F hang off this group.

Common mistake

Treating directed adjacency as two-way. An edge *(u, v)* makes *v* a neighbor of *u*, but without *(v, u)*, *u* is not a neighbor of *v*. That does not make *u* unreachable from *v*: an indirect path may exist.

14.1.5 Paths and cycles

Definition 14.4

A **path** is a vertex sequence $v_0, v_1, \ldots, v_k$ in which each (v_i, v_{i+1}) is an edge. A path repeating no vertex is a **simple path**. A path that ends where it starts is a **cycle**; it is a **simple cycle** if no vertex repeats except the start at the end.

In an unweighted graph a path's **length** is its number of edges: $k + 1$ vertices, length k. A path that follows the arrows of a directed graph is a **directed path**; such a cycle is a **directed cycle**.

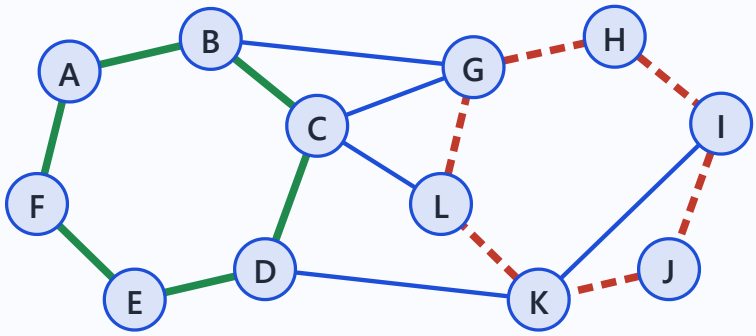


Figure 14.6: Two simple cycles in a graph with twelve vertices: the green A-B-C-D-E-F-A and the dashed red G-H-I-J-K-L-G. The edges B–G, C–G, C–L, and D–K connect them, and the red cycle also has the chord K–I.

Both cycles in Figure 14.6 have length six, and there are others: C → G → L → C uses the edges C–G, G–L, and L–C. The number of cycles grows fast with the number of edges.

In a tree, exactly one simple path joins any two nodes. In a graph there may be several, and two different simple paths together form a cycle. Istanbul → Ankara → Izmir → Istanbul is a simple cycle of flights: no city is visited twice. Istanbul → Ankara → Izmir → Ankara → Istanbul passes Ankara twice, so it is a cycle but not a simple one.

14.1.6 Two example graphs

Here are the concepts at work on a small undirected graph and a small directed one.

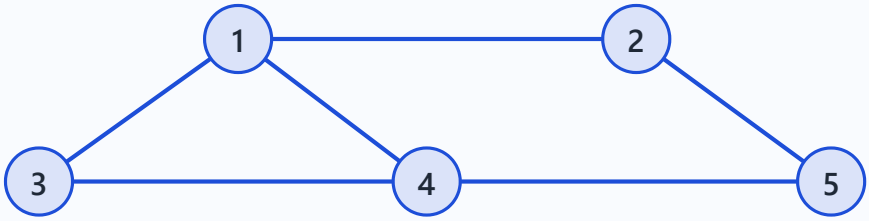


Figure 14.7: The undirected example graph: $V = \{1, 2, 3, 4, 5\}$, $E = \{(1,2), (1,3), (1,4), (2,5), (3,4), (4,5)\}$.

Example 14.2

For the graph of Figure 14.7:

1. **Adjacency.** 1 and 2 are neighbors because *(1,2)* is an edge; 2 and 3 are not.
2. **Degrees.** 1 has neighbors 2, 3, 4 and 4 has 1, 3, 5, so $deg(1) = deg(4) = 3$; 2, 3, and 5 have two each. The sum $3 + 2 + 2 + 3 + 2 = 12 = 2 \cdot 6$ confirms the lemma.
3. **Kinds of path.** The table below classifies four sequences.

Sequence	Kind	Reason
$1 \to 2 \to 5$	simple path	edges 1–2, 2–5 exist; no repeats
$1 \to 3 \to 4 \to 1$	simple cycle	ends at 1; no repeats between
$1 \to 3 \to 4 \to 1 \to 2$	non-simple path	all edges exist; 1 appears twice
$1 \to 3 \to 4 \to 1 \to 4 \to 1$	non-simple cycle	ends at 1; 1 and 4 repeat between

Table 14.2: Four vertex sequences in the graph of Figure 14.7.

A sequence is a path if each consecutive pair is an edge; $2 \to 5 \to 3$ is not, since 5–3 is missing. Only then do we check simplicity: does a vertex repeat?

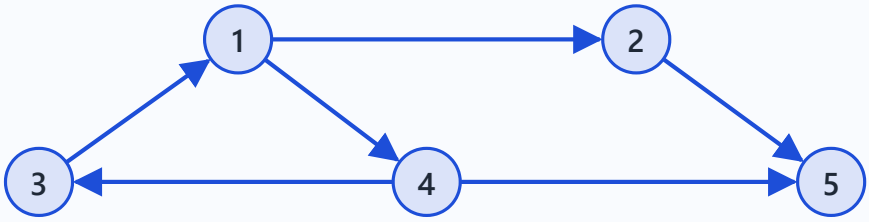


Figure 14.8: The directed example graph: $V = \{1, 2, 3, 4, 5\}$, $E = \{(1,2), (1,4), (2,5), (4,5), (3,1), (4,3)\}$.

Example 14.3

In the directed graph of Figure 14.8:

- 2 is a neighbor of 1 (arrow $1 \to 2$), but 1 is not a neighbor of 2 (no arrow $2 \to 1$).
- $1 \to 2 \to 5$ is a directed path.
- $1 \to 4 \to 3 \to 1$ is a directed cycle (arrows $1 \to 4$, $4 \to 3$, $3 \to 1$).
- No arrow leaves 5, so no other vertex is reachable from it.

14.1.7 Connected and complete graphs

Definition 14.5

An undirected graph is **connected** if there is a path between every two distinct vertices. A largest group of vertices that can all reach one another is a **connected component**. A graph is **complete** if every two distinct vertices are joined directly by an edge; the complete graph on n vertices is written K_n.

Connectivity asks for a *path*, completeness for an *edge*: a complete graph is connected, not necessarily vice versa. “Largest” matters: a component includes every vertex that can reach any of its members. Components split the vertices into disjoint groups; a connected graph has one.

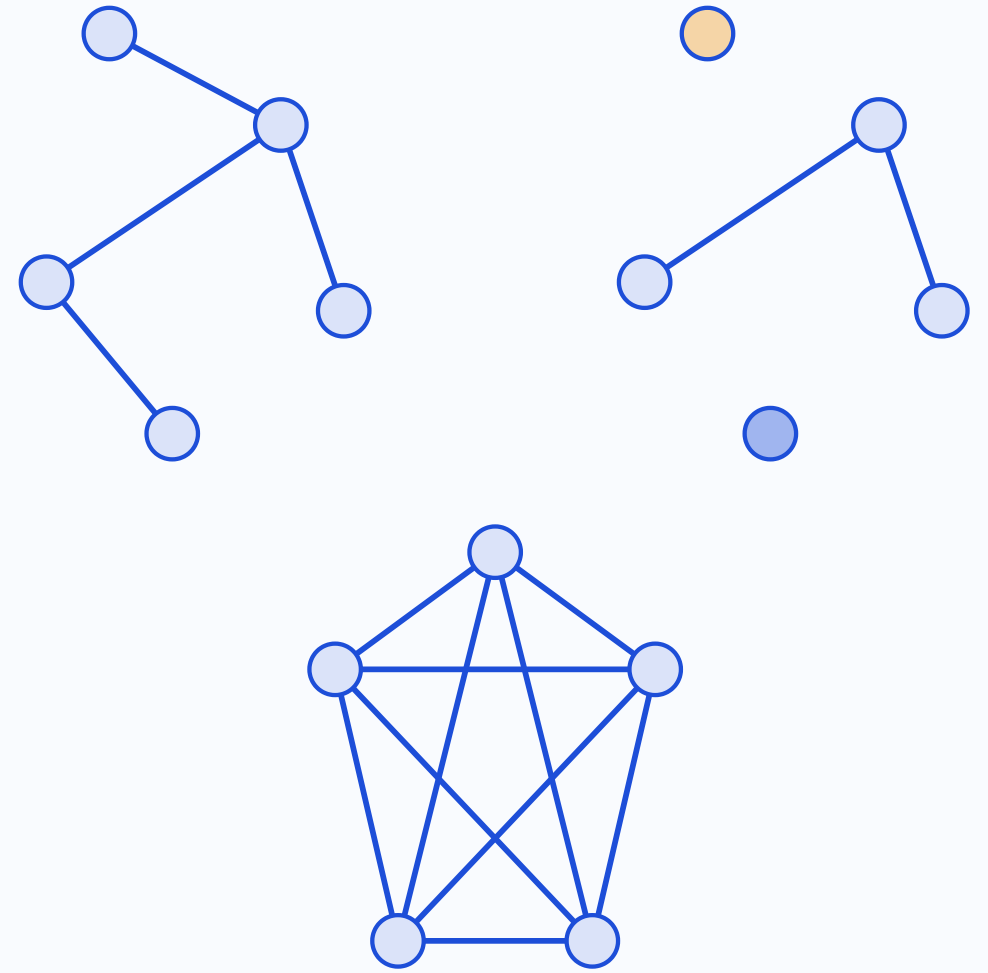


Figure 14.9: Left to right: a connected graph, five vertices held together by four edges (1 component); a disconnected graph made of two vertices joined by an edge and two isolated vertices (3 components); the complete graph K_5 (10 edges).

The edge count has two bounds. In a simple undirected graph (no self-loops or parallel edges) each vertex pair has at most one edge, and there are $n(n-1)/2$ pairs: n choices, then $n-1$, halved because each pair is counted twice. So $m \le n(n-1)/2$, with equality only when complete: K_5 has $5 \cdot 4 / 2 = 10$ edges, the complete graph on 10 vertices $10 \cdot 9 / 2 = 45$.

A connected graph has $m \ge n - 1$. Add edges one by one to an empty graph: it starts with n components, and each edge either merges two or falls inside one, so reaching one component takes at least $n - 1$ edges. A connected graph with exactly $n - 1$ edges is a tree, as we will see again with spanning trees (14.5.1).

These bounds separate **sparse** from **dense** graphs: $m \approx n$ is sparse, $m \approx n^2$ dense. Road and social networks are sparse; small groups where everyone knows everyone are dense.

14.1.8 Weighted graphs

Often a connection has a cost: a road's length, a link's latency, a pipeline's price, a line's capacity.

Definition 14.6

In a **weighted graph**, each edge is assigned a number called its **weight**. The **length** or **weight** of a path is the sum of the weights of its edges. A weighted graph may be undirected or directed.

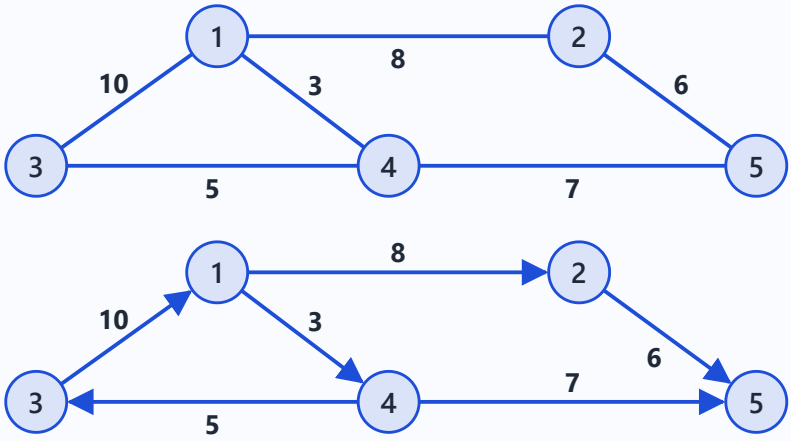


Figure 14.10: A weighted undirected graph (top) and a weighted directed graph (bottom) on the same five vertices. The undirected edges are 1–2: 8, 1–3: 10, 1–4: 3, 2–5: 6, 3–4: 5, 4–5: 7; the arrows are 1 → 2: 8, 1 → 4: 3, 2 → 5: 6, 4 → 5: 7, 3 → 1: 10, 4 → 3: 5.

Example 14.4

In the undirected graph of Figure 14.10, the path 3 → 4 → 1 has length $5 + 3 = 8$; in the directed graph, 1 → 4 → 3 has length $3 + 5 = 8$.

The direct edge 1–3 weighs 10, yet 1 → 4 → 3 costs 8. A direct edge need not be the shortest path; more edges can be shorter if they are light.

Hence shortest-path algorithms. Without weights the shortest path has the fewest edges; with weights, edge counts mislead and the weight sum decides. An unweighted graph is a weighted one with all weights 1.

14.1.9 The graph ADT

A graph can first be defined as an **abstract data type** (ADT): which operations exist and what they do, not how memory carries them out. The `Graph` interface of Goodrich, Tamassia, and Goldwasser has these basic operations.

Operation	Description
`numVertices()`, `numEdges()`	Number of vertices n, number of edges m
`vertices()`, `edges()`	Iteration over all vertices / all edges
`getEdge(u, v)`	Returns the edge from u to v if there is one, otherwise `null`
`endVertices(e)`, `opposite(v, e)`	The two endpoints of an edge / the endpoint of e other than v
`outDegree(v)`, `inDegree(v)`	Out-degree / in-degree (equal in an undirected graph)
`outgoingEdges(v)`, `incomingEdges(v)`	Edges leaving v / entering v
`insertVertex(x)`, `insertEdge(u, v, x)`	Adds a new vertex / a new edge (x: the value stored)
`removeVertex(v)`, `removeEdge(e)`	Removes a vertex with its incident edges / removes an edge

Table 14.3: Basic operations of the graph ADT (V: vertex, E: edge).

Algorithms lean on two of these: `getEdge(u, v)` (are u and v adjacent?) and `outgoingEdges(v)` (who are v's neighbors?). BFS, DFS, Dijkstra, and Prim call the second once per vertex, so how fast a representation lists neighbors sets their running time.

`removeVertex(v)` removes the incident edges for consistency: an edge without an endpoint is meaningless. The cost of each operation depends on the representation.

14.2 Graph representations

A graph can be stored in three forms, each speeding up some operations and slowing others. We take them in turn, compare their costs, and trace how Java builds an adjacency list and matrix.

14.2.1 Three representations; the edge list

The three basic ways to store a graph are:

- **Edge list**: all edges in a single list, such as `[(A,B), (A,E), …]`.
- **Adjacency list**: for each vertex, a list of its neighbors, such as `V[i] = {j, k, l}`.
- **Adjacency matrix**: an $n \times n$ array whose cell `M[i][j]` records whether there is an edge from i to j.

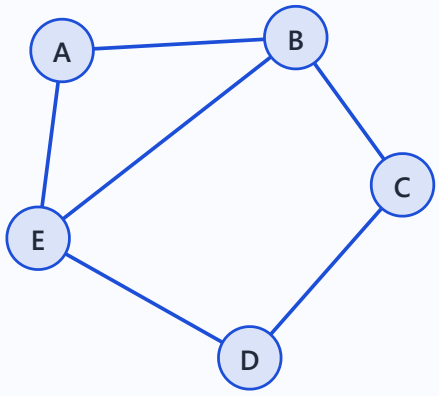


Figure 14.11: The five-vertex undirected graph used to compare the representations, with edges A–B, A–E, B–C, B–E, C–D, and D–E.

The edge list is the simplest: for Figure 14.11 it is `(A, B) (A, E) (B, C) (B, E) (C, D) (D, E)`. A separate vertex list keeps edgeless vertices, for $O(n + m)$ memory. Adding an edge appends one item: $O(1)$.

Queries are the weakness. Testing adjacency means searching the unordered list for (u, v) or (v, u), $O(m)$ in the worst case, and finding u's neighbors means scanning it all, since u's edges can be anywhere. Traversals ask this for every vertex, so they cost $O(n \cdot m)$. Kruskal's algorithm, which sorts edges by weight and never asks for neighbors, is where the edge list shines (14.5.3).

14.2.2 The adjacency matrix

Definition 14.7

The **adjacency matrix** of a graph whose vertices are numbered 0 to $n - 1$ is an $n \times n$ array `matrix` with `matrix[i][j] = 1` if there is an edge from i to j and 0 otherwise. In a weighted graph the cell holds the edge's weight, and a missing edge is marked with ∞ or some special value meaning the same.

An undirected edge (i, j) goes into both `matrix[i][j]` and `matrix[j][i]`, so the matrix is **symmetric** about the diagonal: `matrix[i][j] == matrix[j][i]`. A directed graph's matrix is not.

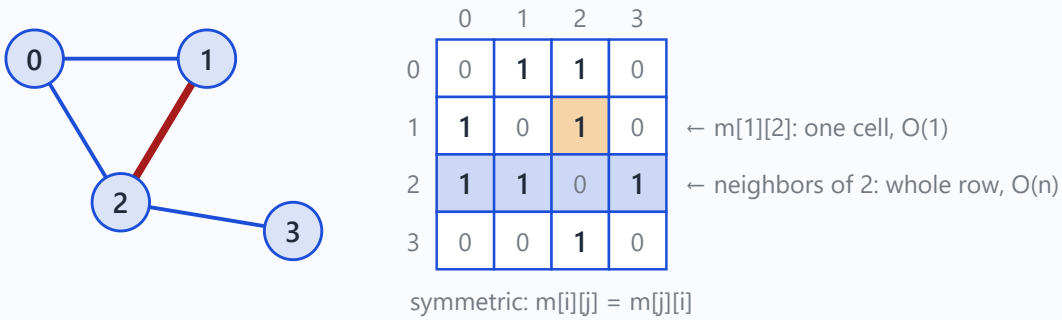


Figure 14.12: A four-vertex graph (edges 0–1, 0–2, 1–2, 2–3) and its adjacency matrix. Checking `m[1][2]` takes a single access, *O(1)*; finding the neighbors of 2 means reading all of row 2, *O(n)*.

The matrix's strength is direct access: a Java array cell is read or written in constant time, so three operations are *O(1)*:

- Testing for edge *i*–*j* reads `matrix[i][j]`.
- Adding an edge writes 1 into one cell (two if undirected).
- Removing it writes 0 there.

It has two weaknesses. Listing neighbors scans all n cells of a row, *O(n)* even for a single neighbor. And the matrix takes n^2 cells regardless of edges; in a sparse graph nearly all are 0, memory spent on edges that do not exist.

Example 14.5

A road network of a million vertices ($n = 10^6$) needs $n^2 = 10^{12}$ matrix cells, about 125 GB even at one bit each. But three or four roads meet at an intersection, so there are a few million edges: a few filled cells per million.

14.2.3 Adjacency matrices: three examples

Here are the matrices of three graphs on the vertices A–E: undirected, directed, and weighted. Rows and columns run A, B, C, D, E.

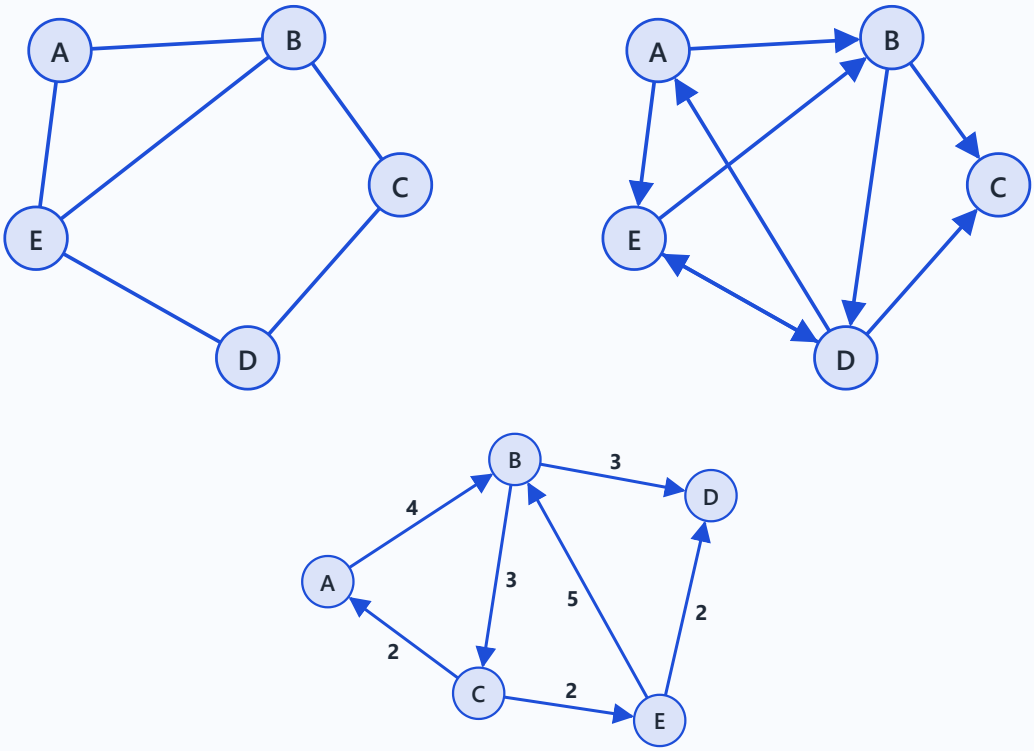


Figure 14.13: Left to right: the undirected graph (the graph of Figure 14.11); a directed graph with nine arrows (A → B, A → E, B → C, B → D, D → A, D → C, D → E, E → B, E → D); a weighted directed graph (A → B: 4, B → C: 3, B → D: 3, C → A: 2, C → E: 2, E → B: 5, E → D: 2).

	A	B	C	D	E
A	0	1	0	0	1
B	1	0	1	0	1
C	0	1	0	1	0
D	0	0	1	0	1
E	1	1	0	1	0

Table 14.4: Adjacency matrix of the undirected graph.

Each undirected edge fills two cells: six edges, twelve 1s, symmetric about the diagonal (row A, column E and row E, column A are both 1). Row sums are degrees: row B has three 1s (A, C, E), so *deg(B) = 3*. Without self-loops, the diagonal is 0.

	A	B	C	D	E
A	0	1	0	0	1
B	0	0	1	1	0
C	0	0	0	0	0
D	1	0	1	0	1

	A	B	C	D	E
E	0	1	0	1	0

Table 14.5: Adjacency matrix of the directed graph (row = source, column = destination).

Each arrow fills one cell: nine arrows, nine 1s. Rows are sources and columns destinations, so there is no symmetry: A → B sets `m[A][B] = 1`, but `m[B][A] = 0`. Row sums are out-degrees, column sums in-degrees: D's out-degree is 3; row C is all 0s, since nothing leaves C; column C has two 1s (from B and D), so C's in-degree is 2.

	A	B	C	D	E
A	0	4	∞	∞	∞
B	∞	0	3	3	∞
C	2	∞	0	∞	2
D	∞	∞	∞	0	∞
E	∞	5	∞	2	0

Table 14.6: Adjacency matrix of the weighted directed graph (∞ for no edge, 0 on the diagonal).

Weighted cells hold weights: the 4 at row A, column B weighs A → B. Missing edges hold ∞; the diagonal holds 0, a vertex's distance to itself. This suits shortest paths: ∞ means “no direct route” and behaves in sums, since ∞ plus anything is ∞.

Common mistake

Writing 0 for missing edges in a weighted matrix. A 0 looks like a real edge of weight 0, so the algorithm takes a free ride that does not exist. Java starts every `int[][]` cell at 0, so the mistake happens by default; fill edgeless cells with a value such as `Integer.MAX_VALUE`. Diagonal 0s are correct.

14.2.4 The adjacency list

Definition 14.8

An **adjacency list** keeps one list per vertex, n lists in all. `list[i]` holds the neighbors of i (with their weights, in a weighted graph). The lists are usually linked lists or dynamic arrays.

An undirected edge *(v, w)* goes into two lists: *w* into *v*'s and *v* into *w*'s. A directed edge goes only into its source's list. Since only existing edges are stored, memory is *n* list heads plus the entries: *2m* entries if undirected, *m* if directed, *O(n + m)* either way.

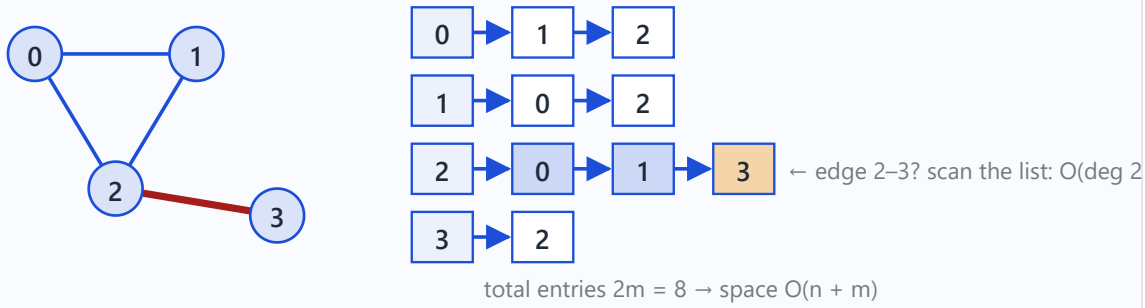


Figure 14.14: Adjacency list of the four-vertex graph of Figure 14.12: 0: 1, 2 · 1: 0, 2 · 2: 0, 1, 3 · 3: 2. To ask whether edge 2–3 exists, scan 2's list: *O(deg 2)*. There are *2m = 8* entries in all.

Lists are strong at traversal: visiting a vertex's neighbors reads its list in *O(deg(v))*, never touching non-neighbors as the matrix does. Adding an edge appends: amortized *O(1)* for a dynamic array, *O(1)* for a linked list.

They are weak at edge queries. Unless sorted, *u*'s list must be scanned for *v*: *O(deg(u))*, or *O(min(deg u, deg v))* if an undirected graph scans the shorter list. For high-degree vertices this is slow next to the matrix's *O(1)*.

14.2.5 Adjacency lists: three examples

Here are the adjacency lists of the three graphs of Figure 14.13, neighbors in alphabetical order.

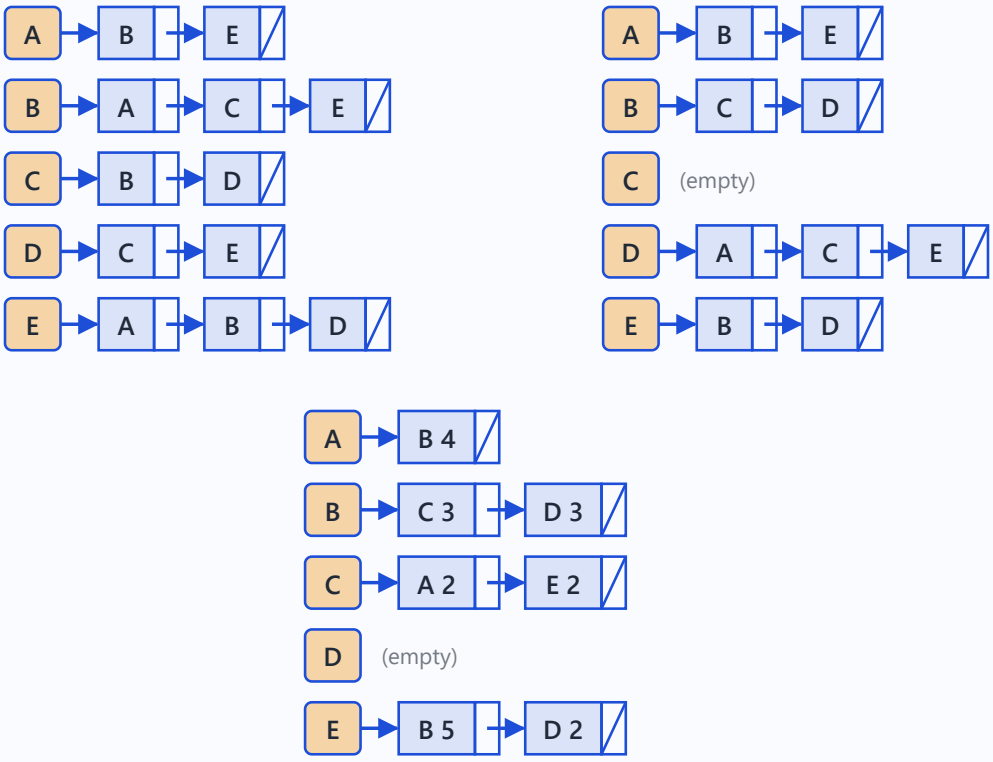


Figure 14.15: Left to right: lists of the undirected graph (A: B, E · B: A, C, E · C: B, D · D: C, E · E: A, B, D); of the directed graph (A: B, E · B: C, D · C empty · D: A, C, E · E: B, D); of the weighted directed graph (A: (B, 4) · B: (C, 3), (D, 3) · C: (A, 2), (E, 2) · D empty · E: (B, 5), (D, 2)).

- **Undirected.** Six edges make 12 entries, each edge written twice. A list's length is the vertex's degree: B's list has three items, and *deg(B) = 3*.
- **Directed.** Only outgoing edges: nine arrows, nine entries; C's list is empty. Length is the out-degree; in-degrees need a scan of every list, a drawback for directed graphs.
- **Weighted.** Entries are (neighbor, weight) pairs; B's list is `(C, 3), (D, 3)`. Java uses an `int[]{v, w}` or a small `Edge` class.

14.2.6 What each representation costs

Operation	Edge list	Adjacency list	Adjacency matrix
Memory	O(n + m)	O(n + m)	O(n^2)
Edge u–v? (`getEdge`)	O(m)	undirected: O(min(deg u, deg v)); directed u → v: O(outDeg u)	**O(1)**
Visit v's neighbors	O(m)	**O(deg v)**	O(n)
Add edge	O(1)	O(1)	O(1)

Table 14.7: Memory and time costs of the three representations (*n* vertices, *m* edges).

Choose by the graph's structure and its most frequent operation:

- **Sparse graphs** ($m \approx n$), like road and social networks: a matrix's n^2 cells sit nearly empty, while a list keeps memory at $O(n + m)$ and speeds traversal. Every traversal and shortest-path algorithm here runs in or near $O(n + m)$ with lists.
- **Dense graphs** ($m \approx n^2$): few cells are wasted, and frequent edge queries favor the matrix's $O(1)$.

Example 14.6

Which representation suits a road network with $n = 10{,}000$ vertices and $m = 30{,}000$ edges?

1. **Matrix.** $n^2 = 10^8$ cells, of which $2m = 60{,}000$ are filled in an undirected graph: $60{,}000 / 10^8 = 0.06\%$.
2. **List.** n heads and $2m$ entries: $10{,}000 + 60{,}000 = 70{,}000$ items, the same order as $n + m = 40{,}000$, so memory is $O(n + m)$.
3. **Decision.** The graph is sparse ($m = 3n$): choose the adjacency list. Traversal, too, takes $O(n + m)$ with the list and $O(n^2)$ with the matrix.

Common mistake

Confusing the matrix's row scan with `getEdge`. In a matrix, testing for an edge reads one cell, $O(1)$; what costs $O(n)$ is scanning a row for all neighbors. Lists are the other way around: visiting neighbors is cheap, querying one edge relatively expensive.

14.2.7 An adjacency list in Java

With vertices numbered 0 to $n - 1$, an adjacency list is a list of lists: item i of the outer list is the neighbor list of vertex i.

```
class Graph {
    private final int n;                                    // vertices: 0..n-1
    private final List<List<Integer>> adj = new ArrayList<>();
    Graph(int n) {
        this.n = n;
        for (int i = 0; i < n; i++)
            adj.add(new ArrayList<>());
    }
    void addEdge(int u, int v) {
        adj.get(u).add(v);
        adj.get(v).add(u);                    // omit this line in a directed graph
    }
    List<Integer> neighbors(int u) { return adj.get(u); }   // O(1)
    int degree(int u) { return adj.get(u).size(); }
}
```

- `adj` is a `List<List<Integer>>`: one inner `ArrayList` (dynamic array) per vertex.
- The constructor stores n and adds n empty lists in $O(n)$. All vertices are known up front, so `adj.get(u)` is valid for every u.
- `addEdge(u, v)` puts v in u's list and u in v's; a directed graph drops the second line. Like `ArrayList.add`, it is amortized $O(1)$.
- `neighbors(u)` returns the list itself, uncopied, in $O(1)$; iterating over it costs the caller $O(\deg(u))$.
- `degree(u)` is the list's size, $O(1)$ since `ArrayList` stores it.

The matrix version is shorter: `boolean[][] m = new boolean[n][n];` creates it, `m[u][v] = m[v][u] = true;` adds an edge, `m[u][v]` queries. With weights, a list item becomes an `int[]{v, w}` or a small `Edge` class, and the matrix an `int[][]` with `Integer.MAX_VALUE` for missing edges.

14.2.8 Code trace: building an adjacency list

We build a five-vertex graph from six edges, in the order *{0,1}, {0,4}, {1,2}, {1,4}, {2,3}, {3,4}*. The traced code is the constructor, `addEdge`, and the `main` fragment that calls them.

```
Graph(int n) {
  this.n = n;
  for (int i = 0; i < n; i++)
    adj.add(new ArrayList<>());
}
void addEdge(int u, int v) {
  adj.get(u).add(v);
  adj.get(v).add(u);  // not if directed
}
// main:
Graph g = new Graph(5);
int[][] E = {{0,1}, {0,4}, {1,2},
             {1,4}, {2,3}, {3,4}};
for (int[] e : E) g.addEdge(e[0], e[1]);
```

Step	0	1	2	3	4
constructor (`i = 0 … 4`)	[]	[]	[]	[]	[]
`addEdge(0, 1)`	[1]	[0]	[]	[]	[]
`addEdge(0, 4)`	[1, 4]	[0]	[]	[]	[0]
`addEdge(1, 2)`	[1, 4]	[0, 2]	[1]	[]	[0]
`addEdge(1, 4)`	[1, 4]	[0, 2, 4]	[1]	[]	[0, 1]
`addEdge(2, 3)`	[1, 4]	[0, 2, 4]	[1, 3]	[2]	[0, 1]
`addEdge(3, 4)`	[1, 4]	[0, 2, 4]	[1, 3]	[2, 4]	[0, 1, 3]

Table 14.8: Neighbor lists after `new Graph(5)` and each of the six `addEdge` calls.

First the constructor: `new Graph(5)` sets `n = 5`, and the loop runs `i = 0` to `4`, giving each vertex an empty list.

Then the edges. `addEdge(0, 1)` appends 1 to 0's list, then 0 to 1's; every call does the same. Lists keep insertion order: 1's list is `[0, 2, 4]` because 0–1, 1–2, and 1–4 arrived in that order.

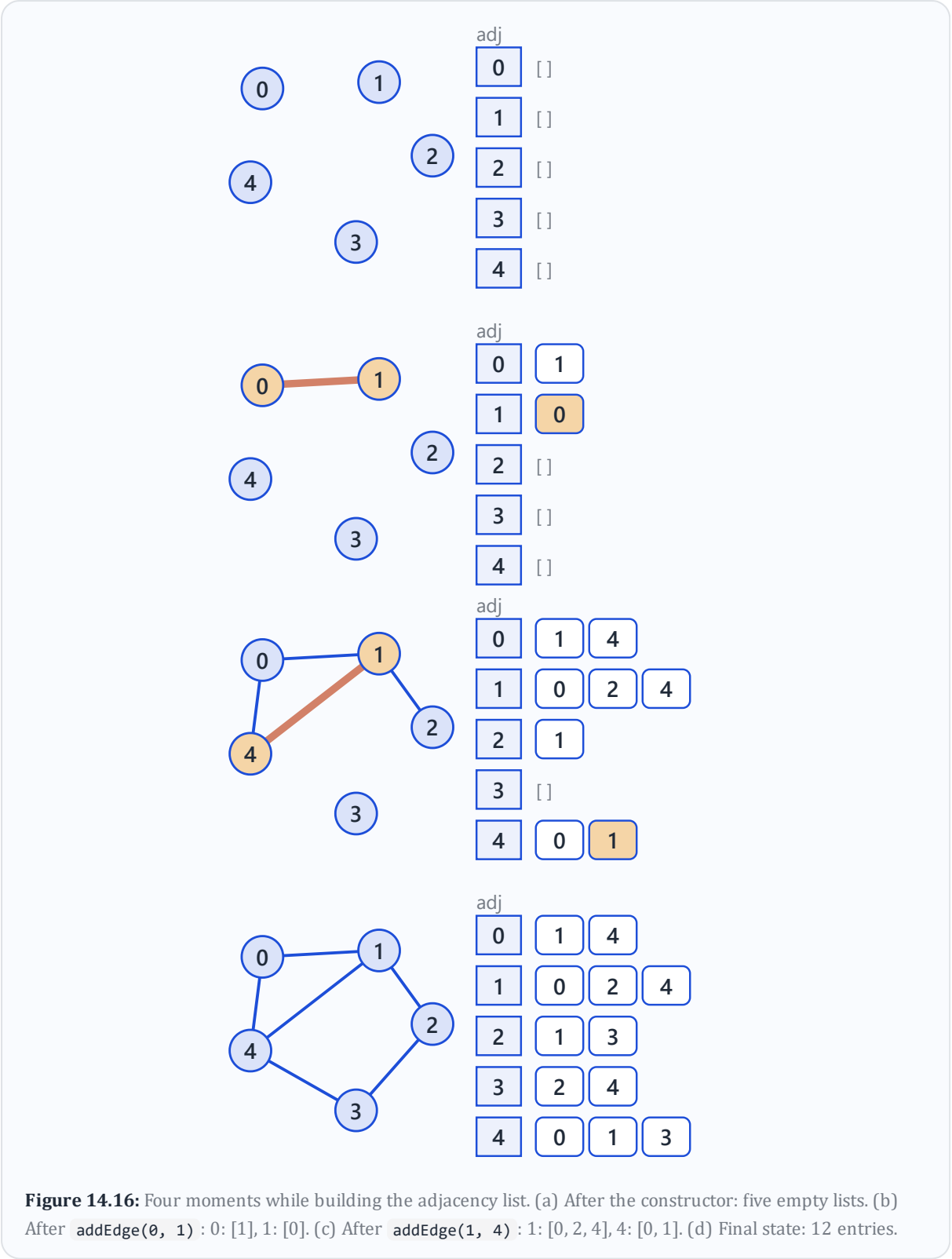


Figure 14.16: Four moments while building the adjacency list. (a) After the constructor: five empty lists. (b) After `addEdge(0, 1)`: 0: [1], 1: [0]. (c) After `addEdge(1, 4)`: 1: [0, 2, 4], 4: [0, 1]. (d) Final state: 12 entries.

Six edges produce $2 \cdot 6 = 12$ entries. The list lengths 2, 3, 2, 2, 3 sum to 12, the handshake lemma as it shows up in code; `degree(1)` is the size of 1's list, 3. Each `addEdge` takes constant time, and memory is $O(n + m)$.

> **Common mistake**
>
> Dropping the second line of `addEdge` in an undirected graph. The graph silently turns directed: 0–1 is in 0's list but not 1's, so a traversal from 1 never reaches 0. No error, just wrong answers. Conversely, adding a one-way relationship both ways invents return routes.

14.2.9 Code trace: building an adjacency matrix

Now the same six edges go into a 5×5 `boolean` matrix.

```
boolean[][] m = new boolean[n][n];
void addEdge(int u, int v) {
  m[u][v] = true;
  m[v][u] = true;  // not in a directed graph
}
// main: the same 6 edges (n = 5)
for (int[] e : E) g.addEdge(e[0], e[1]);
```

`new boolean[n][n]` allocates 25 cells, which Java initializes to `false`: no edges yet. Even creating the matrix takes $O(n^2)$, since every cell must receive its initial value.

Each `addEdge(u, v)` writes two cells. `addEdge(0, 1)` sets `m[0][1] = true` (row 0, column 1) and then the symmetric cell `m[1][0]`. The six edges fill the cell pairs (0,1)–(1,0), (0,4)–(4,0), (1,2)–(2,1), (1,4)–(4,1), (2,3)–(3,2), (3,4)–(4,3).

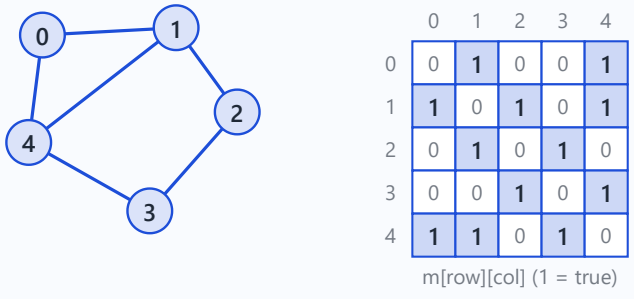


Figure 14.17: The adjacency matrix after the six edges: 12 cells `true`, 13 `false`, symmetric about the diagonal.

	0	1	2	3	4
0	0	1	0	0	1

	0	1	2	3	4
1	1	0	1	0	1
2	0	1	0	1	0
3	0	0	1	0	1
4	1	1	0	1	0

Table 14.9: The final matrix (1 = `true`, 0 = `false`).

Even here the difference shows. “Edge 2–4?” is one access to `m[2][4]` (`false`); the list must scan `[1, 3]`. But 2's neighbors cost five cells in the matrix and two items in the list. Thirteen of 25 cells carry nothing, and the ratio worsens fast as graphs grow and thin out.

14.3 Traversal: BFS and DFS

Most graph algorithms rest on a systematic exploration of the graph: breadth-first or depth-first search. Both visit the same vertices, in orders that suit different problems. Four applications follow: connected components, bipartiteness, topological sorting, and directed cycle detection.

14.3.1 Traversal and visited marks

> **Definition 14.9**
>
> A **traversal** visits every vertex reachable from a starting vertex, each exactly once, in a systematic order.

There are two basic methods:

- **Breadth-first search** (BFS) visits all neighbors of the start, then their neighbors, level by level, using a **queue** (FIFO: first in, first out).
- **Depth-first search** (DFS) goes as deep as it can along one path, backs up at a dead end, and tries another branch, using a **stack** (LIFO: last in, first out) or recursion.

In a tree one path leads from the root to each node, so descending to children never reaches a node twice. A graph may have cycles and several routes to a vertex, so its traversal must record what it has seen, usually in a `boolean[] visited` array indexed by vertex.

Common mistake

Running a tree traversal on a graph unchanged. Without visited marks, a cycle traps the traversal forever; even without cycles, a vertex reached by several routes is processed repeatedly, and the cost can grow exponentially. The mark is part of the algorithm's correctness, not a detail.

14.3.2 Breadth-first search

Drop a stone into water and the ripples spread in rings, near ones first, then farther ones. BFS works the same way: it finds the vertices one edge from the start, then those two edges away, then three. The vertices *k* edges from the start form **level *k***.

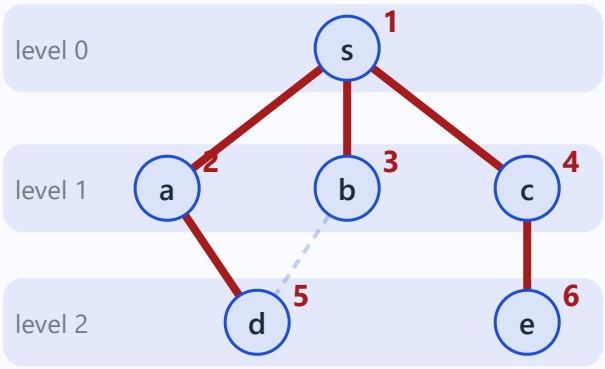


Figure 14.18: BFS proceeds level by level. The number beside each vertex is its visiting order: s (1); a, b, c (2, 3, 4); d, e (5, 6). The bold edges form the BFS tree; the faint dashed edge b–d is not in it, because d was already reached through a.

A queue enforces this order: discovered vertices join the back, and processing takes from the front. All of level 1 enters before any of level 2, so all of level 1 is processed first.

The edges by which BFS discovers vertices form the **BFS tree**. In Figure 14.18, d is reachable from a or b; a comes first, so d enters along a–d and b–d stays out. Tree paths from the root have the fewest possible edges, the basis of unweighted shortest paths (14.4.1).

BFS finds unweighted shortest paths and discovers near vertices first: “friends of friends” is level 2, and a web crawler scans a site layer by layer. Its weakness is memory: a whole level may wait in the queue, which grows large in wide graphs.

14.3.3 The BFS algorithm and code

First the pseudocode, then the Java.

```
BFS(start):
  queue.add(start)
  mark start visited
  while queue is not empty:
    w = queue.remove()
    for each unvisited
        neighbor u of w:
      mark u visited
      queue.add(u)
```

```
void bfs(int s) {
  boolean[] visited = new boolean[n];
  Queue<Integer> q = new ArrayDeque<>();
  visited[s] = true;  q.add(s);
  while (!q.isEmpty()) {
    int w = q.poll();
    System.out.print(w + " ");
    for (int u : adj.get(w))
      if (!visited[u]) {
        visited[u] = true;
        q.add(u);
      }
  }
}
```

The method belongs to the `Graph` class of 14.2.7 and uses its fields `n` and `adj`:

- `boolean[] visited = new boolean[n];` allocates one mark per vertex, all `false`.
- `Queue<Integer> q = new ArrayDeque<>();` creates an empty queue; `add` appends at the back, `poll` removes the front, both *O(1)*.
- `visited[s] = true; q.add(s);` marks and enqueues the start.
- Until the queue empties, each round removes and prints the front vertex `w`.
- The inner loop marks and enqueues each unvisited neighbor `u` of `w`.

The subtle point is *when* to mark: on **entering** the queue, not on leaving. A vertex on several lists then enters once; marking on removal would let another neighbor enqueue it again while it waits.

Common mistake

Marking on dequeue. On the triangle 0–1, 0–2, 1–2, such a BFS from 0 prints `0 1 2 2`: removing 1 enqueues the still unmarked 2 a second time. In dense graphs the queue grows to *O(m)* instead of *O(n)*.

Cost. Each vertex enters and leaves the queue once, $O(n)$, and its list is scanned once; the lists total $2m$ entries, $O(m)$. So BFS takes $O(n + m)$ with lists and $O(n^2)$ with a matrix, whose rows have n cells. *visited* and the queue take $O(n)$ extra memory.

14.3.4 Code trace: bfs(A)

We trace `bfs(A)` on a graph with nine vertices, numbered A = 0, B = 1, ..., I = 8, and 11 edges. Neighbor lists are alphabetical: A: B, F, I · B: A, C, E · C: B, D, E · D: C, G, H · E: B, C, G · F: A, G · G: D, E, F · H: D · I: A.

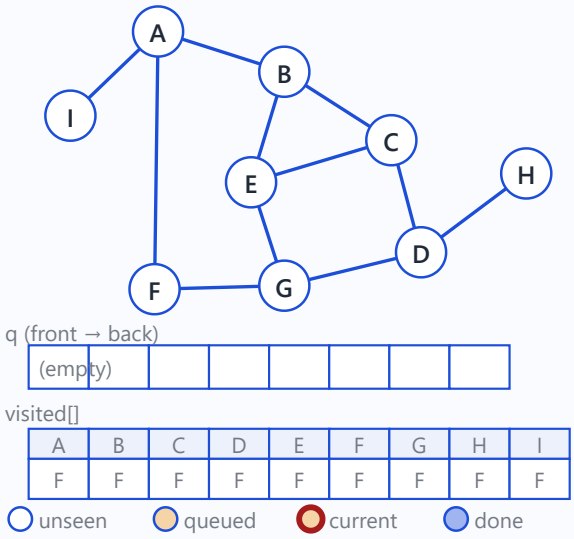


Figure 14.19: The graph used in the traversal examples, in its initial state: the queue is empty and no vertex is visited. The strip below shows each vertex's `visited[]` entry.

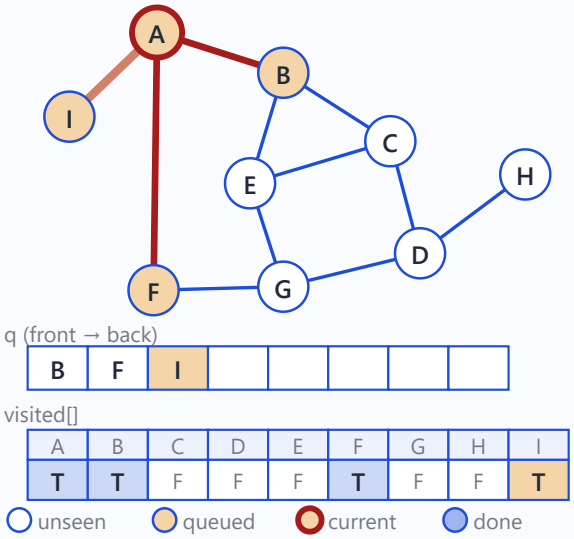
Round	Removed `w` (printed)	Neighbors scanned	Marked and enqueued	Queue (end of round, front to back)
start	—	—	A	[A]
1	A	B, F, I	B, F, I (level 1)	[B, F, I]
2	B	A, C, E	C, E (level 2)	[F, I, C, E]
3	F	A, G	G (level 2)	[I, C, E, G]
4	I	A	—	[C, E, G]
5	C	B, D, E	D (level 3)	[E, G, D]
6	E	B, C, G	—	[G, D]
7	G	D, E, F	—	[D]
8	D	C, G, H	H (level 4)	[H]
9	H	D	—	[]

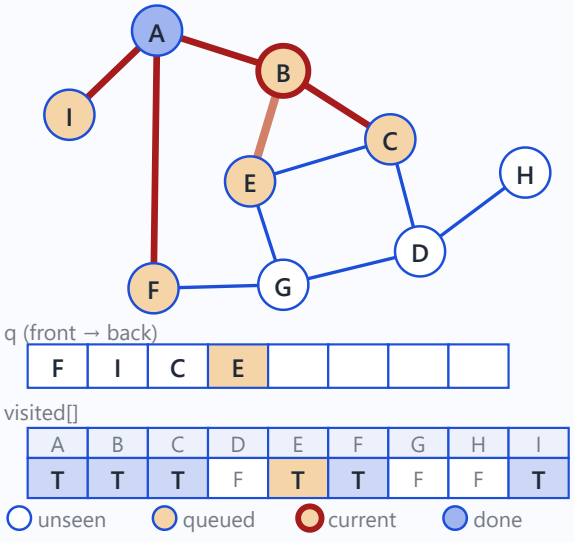
Table 14.10: Trace of `bfs(A)`. Each row is one round of the outer loop.

A is marked and enqueued. Round 1 removes and prints A and enqueues its unvisited neighbors B, F, I: level 1.

Round 2 removes B. Its neighbor A is marked and skipped (without the mark, A would re-enter the queue); C and E join the back, giving `[F, I, C, E]`, level 1 in front, level 2 behind. The queue keeps levels apart by itself.

Round 3 removes F and adds G; round 4 removes I and adds nothing, ending level 1. Round 5 removes C and adds D (level 3). E and G find every neighbor marked. Round 8 has D add H (level 4), and removing H empties the queue.





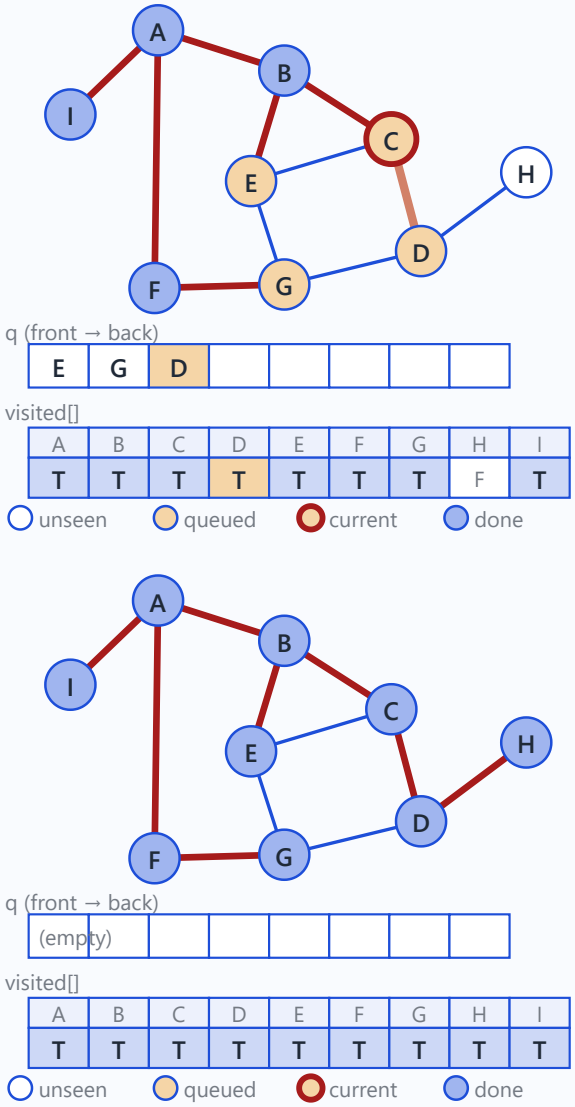


Figure 14.20: Four moments of `bfs(A)`. (a) A processed; queue B, F, I. (b) B processed; queue F, I, C, E. (c) C processed; queue E, G, D. (d) Final state: all vertices processed, bold edges forming the BFS tree.

The output `A B F I C E G D H` shows the levels as consecutive blocks: {A}, {B, F, I}, {C, E, G}, {D}, {H}. The BFS tree has the eight edges A–B, A–F, A–I, B–C, B–E, F–G, C–D, D–H, as nine vertices require. Each vertex was enqueued once; the scans covered $2m = 22$ entries.

14.3.5 Depth-first search

In a maze, you walk a corridor to its end; at a dead end you return to the last fork and try a new corridor. That is DFS: it goes as deep as it can, and at a vertex with no unvisited neighbors it **backtracks** to try another branch.

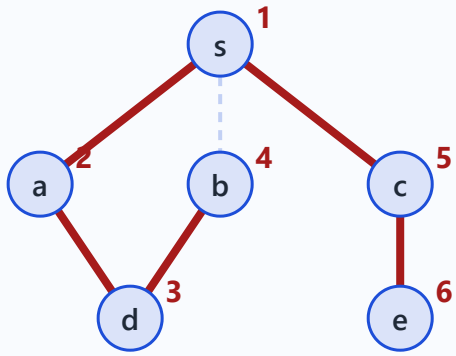


Figure 14.21: DFS goes deep along one path. Numbers give the visiting order: s (1), a (2), d (3), b (4); b is a dead end, so the search backs up; c (5), e (6). Bold edges form the DFS tree. Edge s–b is not in it, because b was reached through d.

Compare Figure 14.18: same graph, different order. BFS visits all of s's neighbors a, b, c next; DFS descends s, a, d, b, finds b's neighbors all seen, backs up to s, and descends to c, then e.

DFS stores only the current path, so on wide, shallow graphs it needs less memory than BFS, and it suits problems where a path is followed to its end (mazes, game trees). But its paths need not be shortest: it reached b by s → a → d → b, three edges, though s–b exists. Its main uses are mazes, cycle detection, connected components, and topological sorting of DAGs.

14.3.6 DFS code: recursive and with an explicit stack

The shortest way to write DFS is recursively.

```
void dfs(int v, boolean[] visited) {
  visited[v] = true;
  System.out.print(v + " ");
  for (int u : adj.get(v))
    if (!visited[u])
      dfs(u, visited);
}
```

The method marks and prints `v`, then calls itself on each unvisited neighbor in turn. The loop waits for each call to return, so everything below the first neighbor is finished before the second: “deepest first.” When the neighbors run out, the method returns and the caller resumes; that return is backtracking.

Recursion keeps the stack for us as the **call stack**: the active `dfs` frames, in order, are the path from the start to the current vertex. An explicit stack works too:

```
void dfsStack(int s) {
  boolean[] visited = new boolean[n];
  Deque<Integer> stack = new ArrayDeque<>();
  stack.push(s);  visited[s] = true;
  while (!stack.isEmpty()) {
    int v = stack.peek();
    Integer next = null;
    for (int u : adj.get(v))
      if (!visited[u]) { next = u; break; }
    if (next == null) stack.pop(); // back up
    else { visited[next] = true;
           stack.push(next); }
  }
}
```

This imitates the recursion step by step. The top (`peek`) is the current vertex; each round seeks its *first* unvisited neighbor. Found, it is marked and pushed, like a recursive call; not found (`next == null`), the top is popped, like a return. The `break` makes the search descend at the first suitable neighbor.

An explicit stack also avoids `StackOverflowError` : Java's call stack is limited, and recursive DFS on a path of hundreds of thousands of vertices can overflow it, while an explicit stack lives on the heap.

Cost. Recursive DFS calls `dfs` once per vertex and walks each list once: $O(n + m)$. The explicit-stack version rescans a list *from the start* at every `peek` , skipping marked neighbors again each time a vertex resurfaces, at most its degree plus one times; the worst case approaches $\Sigma\ deg(v)^2$. A per-list "where I left off" index restores $O(n + m)$. The code favors simplicity; for small degrees the difference is negligible.

14.3.7 Code trace: recursive dfs(A)

We trace `dfs(A, visited)` on the graph of Figure 14.19, neighbors again in alphabetical order.

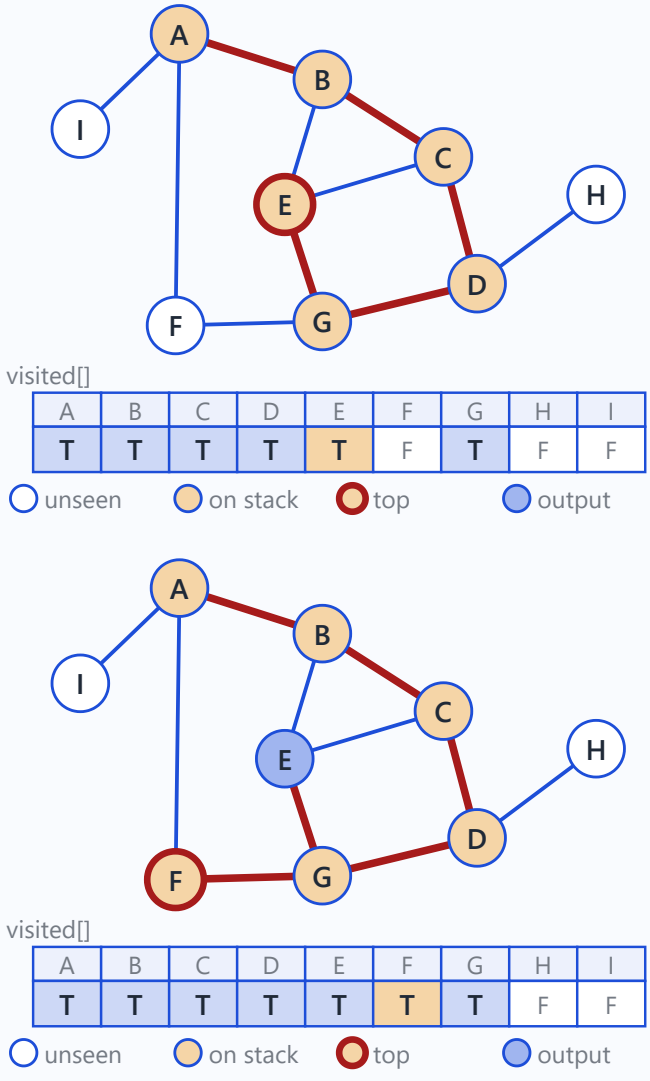
Step	Event	Call stack	Printed (so far)
1	`dfs(A)` : A is marked	A	A
2	A's neighbor B is new → `dfs(B)`	A B	A B
3	B: A marked, skip; C new → `dfs(C)`	A B C	A B C
4	C: skip B; D new → `dfs(D)`	A B C D	A B C D
5	D: skip C; G new → `dfs(G)`	A B C D G	A B C D G
6	G: skip D; E new → `dfs(E)`	A B C D G E	A B C D G E
7	E: B, C, G marked → `dfs(E)` returns	A B C D G	—
8	G: F new → `dfs(F)`	A B C D G F	A B C D G E F
9	F: A, G marked → returns; G has no neighbors left → returns	A B C D	—

Step	Event	Call stack	Printed (so far)
10	D: H new → `dfs(H)`	A B C D H	A B C D G E F H
11	H: skip D → returns; D done → returns	A B C	—
12	C: E marked → returns; B: E marked → returns	A	—
13	A: F marked, skip; I new → `dfs(I)`	A I	A B C D G E F H I
14	I: skip A → returns; A done → returns	(empty)	—

Table 14.11: Trace of recursive `dfs(A)`. The call stack is written bottom to top.

Steps 1–6 only go forward, the stack growing a frame each step: A → B → C → D → G → E. At E every neighbor is marked, and the first backtrack returns to G, which resumes its list: D and E are done, F is next.

A caller resumes where it left off, not *from the start*. In step 12 the search returns to C, whose list B, D, E has only E left, already marked below G. Each list is walked exactly once.



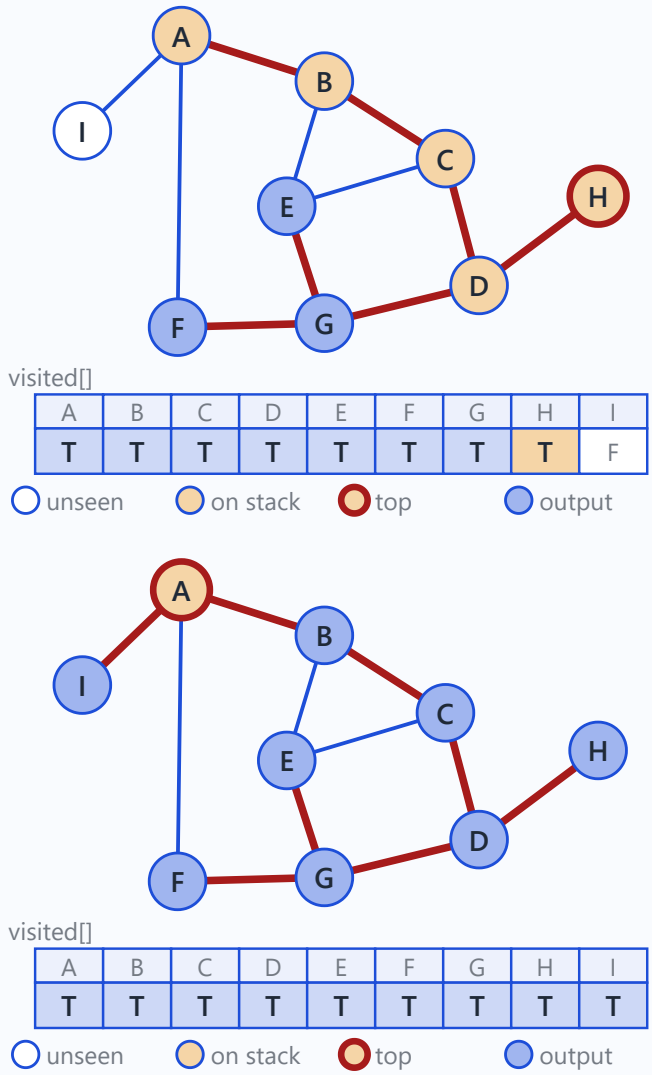


Figure 14.22: Four moments of recursive `dfs(A)`, with the call stack on the right. (a) The deepest point: stack A B C D G E. (b) E has returned, and the search has descended to F below G. (c) G is finished, and the search has descended to H below D. (d) The last step: `dfs(I)` has returned and `dfs(A)` is about to finish; every vertex is visited.

The order is `A B C D G E F H I`; the DFS tree edges are A–B, B–C, C–D, D–G, G–E, G–F, D–H, A–I; the stack peaks at six frames. The DFS tree is longer and thinner than the BFS tree: five edges from root to E, against two.

14.3.8 Code trace: dfsStack(A)

Now the explicit-stack version: each round peeks, seeks the first unvisited neighbor, and pushes or pops.

Round	peek	First unvisited neighbor	Operation	Stack (end of round)
start	—	—	push A	A
1	A	B	push B	A B

Round	peek	First unvisited neighbor	Operation	Stack (end of round)
2	B	C	push C	A B C
3	C	D	push D	A B C D
4	D	G	push G	A B C D G
5	G	E	push E	A B C D G E
6	E	none	pop	A B C D G
7	G	F	push F	A B C D G F
8	F	none	pop	A B C D G
9	G	none	pop	A B C D
10	D	H	push H	A B C D H
11	H	none	pop	A B C D
12	D	none	pop	A B C
13	C	none	pop	A B
14	B	none	pop	A
15	A	I	push I	A I
16	I	none	pop	A
17	A	none	pop	(empty)

Table 14.12: Trace of `dfsStack(A)`. The stack is written bottom to top.

The loop runs 17 rounds: 8 pushes (every vertex but A) and 9 pops (every vertex). Pushes visit vertices in the order `A B C D G E F H I`, as in the recursive version, and the stack always matches the call stack of Table 14.11. One algorithm; only the keeper of the stack differs.

Common mistake

Getting explicit-stack DFS by swapping BFS's queue for a stack: pop a vertex, push all its neighbors. Every vertex is still visited once, but not in DFS order: neighbors come out reversed, and siblings are marked together before the search goes deeper. Hence `peek` and "first neighbor, then `break`".

14.3.9 BFS versus DFS

	BFS	DFS
Helper structure	queue (FIFO)	stack (LIFO) or recursion
Time (adjacency list)	O(n + m)	O(n + m)
Extra memory	O(n): widest level in the queue	O(n): longest path on the stack
Gives shortest paths?	yes (unweighted graphs)	no

	BFS	DFS
Typical use	fewest steps, finding levels	cycle detection, components, mazes, topological sort

Table 14.13: BFS and DFS compared.

Both take *O(n + m)*: each vertex is processed once and each adjacency entry scanned once. An undirected graph has *2m* entries, but the constant vanishes in big-O.

Both need *O(n)* extra memory at worst, in different worst cases: BFS suffers on wide graphs, where many neighbors enter the queue at once; DFS on deep ones, where the stack grows as long as the longest chain.

The traces assume alphabetical lists. Other list orders change the visiting order but not the visited set or the cost: from H, BFS gives `H D C G B E F A I` and DFS `H D C B A F G E I`. And replacing the stack of explicit-stack DFS with a queue, adding all neighbors at once, turns it into BFS: the searches differ in which end of the helper structure they read.

14.3.10 Connected components

A traversal visits only its start's component. To find them all, restart from each unassigned vertex; each restart is a new component.

```
int components(int[] comp) {
  Arrays.fill(comp, -1);
  int count = 0;
  for (int s = 0; s < n; s++)
    if (comp[s] == -1) dfs(s, count++, comp);
  return count;
}
void dfs(int v, int c, int[] comp) {
  comp[v] = c;
  for (int u : adj.get(v))
    if (comp[u] == -1) dfs(u, c, comp);
}
```

`comp` holds component numbers, −1 meaning “unassigned,” and doubles as the visited mark. The outer loop scans vertices 0 to *n* − *1*, calling `dfs(s, count++, comp)` at each −1. The post-increment passes the old value, so the first component is 0. `dfs` writes the number into every vertex it reaches.



Figure 14.23: A six-vertex graph (edges 0–1, 0–2, 3–4) and the result of `components`. The three components are shown in three colors: `comp = [0, 0, 0, 1, 1, 2]`.

`s`	`comp[s]`	**Action**	`count`	`comp` **(after)**
—	—	`Arrays.fill`	0	[−1, −1, −1, −1, −1, −1]
0	−1	`dfs(0, 0)`: colors 0, 1, 2	1	[0, 0, 0, −1, −1, −1]
1	0	skip	1	same
2	0	skip	1	same
3	−1	`dfs(3, 1)`: colors 3, 4	2	[0, 0, 0, 1, 1, −1]
4	1	skip	2	same
5	−1	`dfs(5, 2)`: 5 alone	3	[0, 0, 0, 1, 1, 2]

Table 14.14: Trace of `components` on the graph of Figure 14.23.

Inside `dfs(0, 0)`: 0 is colored, then its neighbor 1, whose only neighbor 0 is done, so the call returns; then 2, likewise. The method returns 3: components *{0, 1, 2}*, *{3, 4}*, *{5}*.

The cost is *O(n + m)*: *n* outer rounds, each vertex colored once with its list walked once, each edge seen once from each end. Afterward, “same component?” is the *O(1)* test `comp[u] == comp[v]`, instead of an *O(n + m)* traversal per query.

14.3.11 Testing whether a graph is bipartite

Definition 14.10

A graph is **bipartite** if its vertices can be split into two groups so that every edge runs *between* the groups; no edge joins two vertices of the same group.

Matching problems produce bipartite graphs: students on one side, projects on the other, an edge meaning “this student wants this project.” Two-shift scheduling and conflict-free grouping look the same.

Think of the groups as two colors, every neighbor of a vertex having the other color. BFS tests this: color the start 0 and each newly discovered vertex the opposite of its discoverer. A colored neighbor with the current vertex's color means the graph is not bipartite.

```
// color[] starts at -1; BFS from s
color[s] = 0;  q.add(s);
while (!q.isEmpty()) {
  int w = q.poll();
  for (int u : adj.get(w))
    if (color[u] == -1) {
      color[u] = 1 - color[w];  q.add(u);
    } else if (color[u] == color[w]) return false;
}
return true;
```

`color` starts at −1 (“uncolored”); `1 - color[w]` flips 0 and 1. It is BFS plus a color check at colored neighbors.

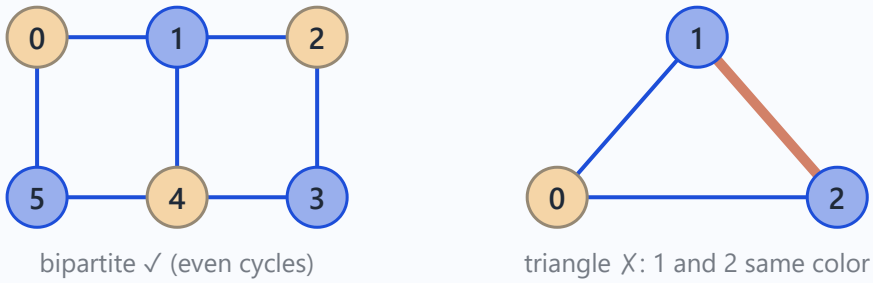


Figure 14.24: Left: a two-colorable six-vertex graph, the cycle 0–1–2–3–4–5–0 plus the edge 1–4. Right: a triangle; colored by BFS from 0, vertices 1 and 2 get the same color, and the edge between them exposes the conflict.

Example 14.7

Left graph. 0 gets color 0; its neighbors 1 and 5 get 1. Processing 1 colors 2 and 4 with 0; processing 5 finds 4 and 0 opposite, as they should be; processing 2 colors 3 with 1, and no later check conflicts. So `color = [0, 1, 0, 1, 0, 1]` : evens in one group, odds in the other. Every cycle is even: 0–1–2–3–4–5–0 has six edges, 0–1–4–5–0 and 1–2–3–4–1 four.

Triangle. 0 gets 0; 1 and 2 get 1. Processing 1 finds neighbor 2 with its own color, and the method returns `false` : three mutually adjacent vertices cannot share two colors.

The general rule: **a graph is bipartite if and only if it has no odd-length cycle.** Colors alternate around a cycle, and returning to the start with its original color requires an even number of edges; so C_5 is not bipartite and C_6 is. The test costs $O(n + m)$, like BFS; a disconnected graph is handled by restarting at each uncolored vertex.

14.3.12 Topological sorting and Kahn's algorithm

A student must take courses in an order where each follows all its prerequisites. With an arrow $u \to v$ when u is a prerequisite of v, this is a directed graph, and the order is a topological order.

Definition 14.11

A **topological sort** of a directed graph arranges its vertices so that for every arrow $u \to v$, u comes first. A directed graph with no directed cycle is a **directed acyclic graph** (DAG). Only DAGs have topological orders.

Why cycles rule it out: given $a \to b \to c \to a$, a must precede b, b precede c, and c precede a, so a precedes itself. Each course on the cycle waits for another, and none can start.

The order need not be unique: courses ready at the same time can go in any order, so the DAG $0 \to 1$, $0 \to 2$, $1 \to 3$, $2 \to 3$ allows $0\ 1\ 2\ 3$ and $0\ 2\ 1\ 3$. Uses include course plans, task dependencies, build order, and workflows.

Kahn's algorithm: a vertex of in-degree 0 has no prerequisites and can go first. Once output, its arrows count as deleted, lowering their heads' in-degrees; vertices reaching 0 become ready and wait in a queue.

```
Kahn(G):
  q = vertices with inDegree 0
  result = []
  while q is not empty:
    u = remove from q
    append u to result
    for each edge (u, v) leaving u:
      inDegree[v]--
      if inDegree[v] == 0:
        add v to q
  if result.size < n:
    there is a cycle
  return result
```

```
List<Integer> kahn(List<List<Integer>> adj) {
  int n = adj.size();
  int[] indeg = new int[n];
  for (List<Integer> l : adj)
    for (int v : l) indeg[v]++;
  Queue<Integer> q = new ArrayDeque<>();
  for (int v = 0; v < n; v++)
    if (indeg[v] == 0) q.add(v);
  List<Integer> result = new ArrayList<>();
  while (!q.isEmpty()) {
    int u = q.poll();
    result.add(u);
    for (int v : adj.get(u))
      if (--indeg[v] == 0) q.add(v);
  }
  return result;  // size < n → cycle
}
```

The Java code has three phases:

1. **Counting.** Nested loops increment `indeg` at each arrow's head, in $O(n + m)$.
2. **Initialization.** Vertices of in-degree 0 are enqueued.
3. **Main loop.** Dequeue u, append it to the result, and run `--indeg[v]` for each arrow's head; the prefix decrement yields the new value, and at 0, v is enqueued.

If the result has n vertices when the queue empties, it is a topological order; fewer means a directed cycle, whose vertices each wait for their predecessor, never reach in-degree 0, and are never enqueued.

14.3.13 Code trace: kahn()

Six courses have the arrows $0 \to 3$, $1 \to 3$, $1 \to 4$, $2 \to 0$, $2 \to 4$, $3 \to 5$, $4 \to 5$; the lists are 0: [3] · 1: [3, 4] · 2: [0, 4] · 3: [5] · 4: [5] · 5: [].

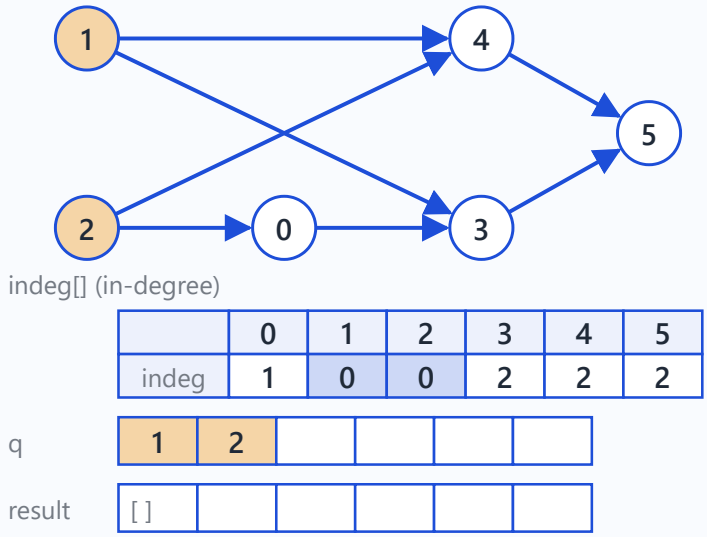


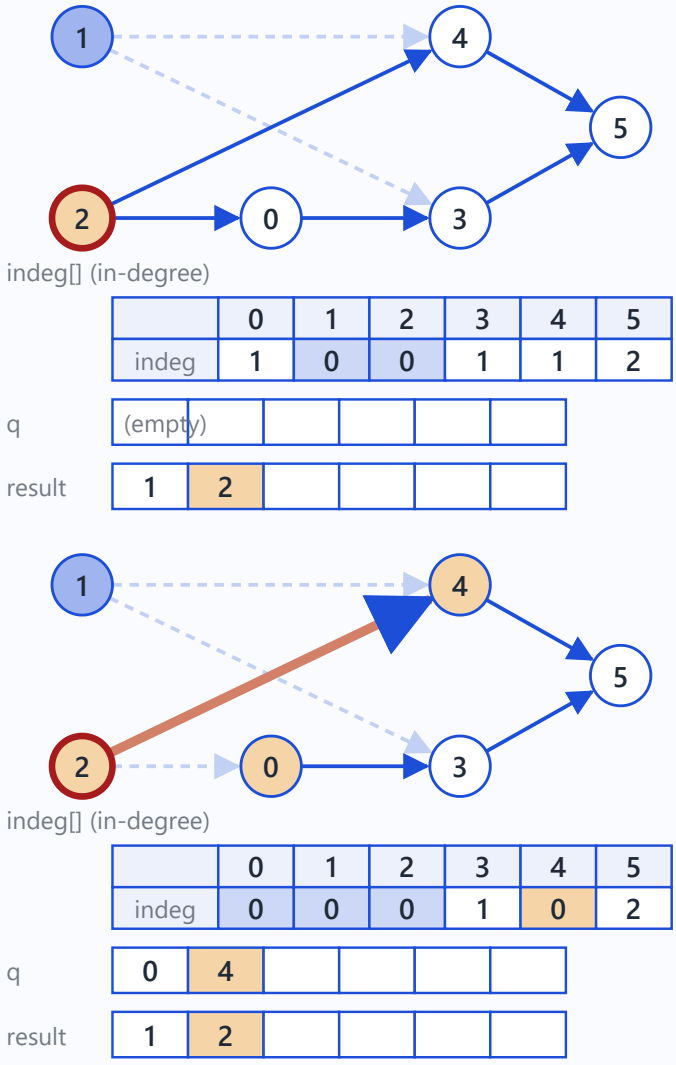
Figure 14.25: The course prerequisite graph after the counting phase: `indeg = [1, 0, 0, 2, 2, 2]`. Courses 1 and 2, which have no prerequisites, are in the queue.

Counting. 0's list sets `indeg[3]` to 1; 1's raises `indeg[3]` to 2 and `indeg[4]` to 1; 2's raises `indeg[0]` to 1 and `indeg[4]` to 2; those of 3 and 4 raise `indeg[5]` to 2. The result is `[1, 0, 0, 2, 2, 2]`, and 1 and 2 enter the queue.

Removed `u`	Deleted arrows and new in-degrees	Enqueued	`indeg` (after)	Queue	`result`
—	—	1, 2	[1, 0, 0, 2, 2, 2]	[1, 2]	[]
1	1 → 3: indeg[3] = 1; 1 → 4: indeg[4] = 1	—	[1, 0, 0, 1, 1, 2]	[2]	[1]
2	2 → 0: indeg[0] = 0; 2 → 4: indeg[4] = 0	0, 4	[0, 0, 0, 1, 0, 2]	[0, 4]	[1, 2]
0	0 → 3: indeg[3] = 0	3	[0, 0, 0, 0, 0, 2]	[4, 3]	[1, 2, 0]
4	4 → 5: indeg[5] = 1	—	[0, 0, 0, 0, 0, 1]	[3]	[1, 2, 0, 4]
3	3 → 5: indeg[5] = 0	5	[0, 0, 0, 0, 0, 0]	[5]	[1, 2, 0, 4, 3]
5	(empty list)	—	[0, 0, 0, 0, 0, 0]	[]	[1, 2, 0, 4, 3, 5]

Table 14.15: Trace of the main loop of `kahn()`.

Removing 1 lowers 3 and 4 but not to 0: 3 waits for 0, 4 for 2. Removing 2 readies 0 and 4; removing 0 readies 3. `indeg[5]` drops to 1 when 4 leaves and to 0 when 3 does.


1
4
5
2
0
3
indeg[] (in-degree)
0 1 2 3 4 5
indeg 1 0 0 1 1 2
q
(empty)
result
1 2
1
4
5
2
0
3
indeg[] (in-degree)
0 1 2 3 4 5
indeg 0 0 0 1 0 2
q
0 4
result
1 2

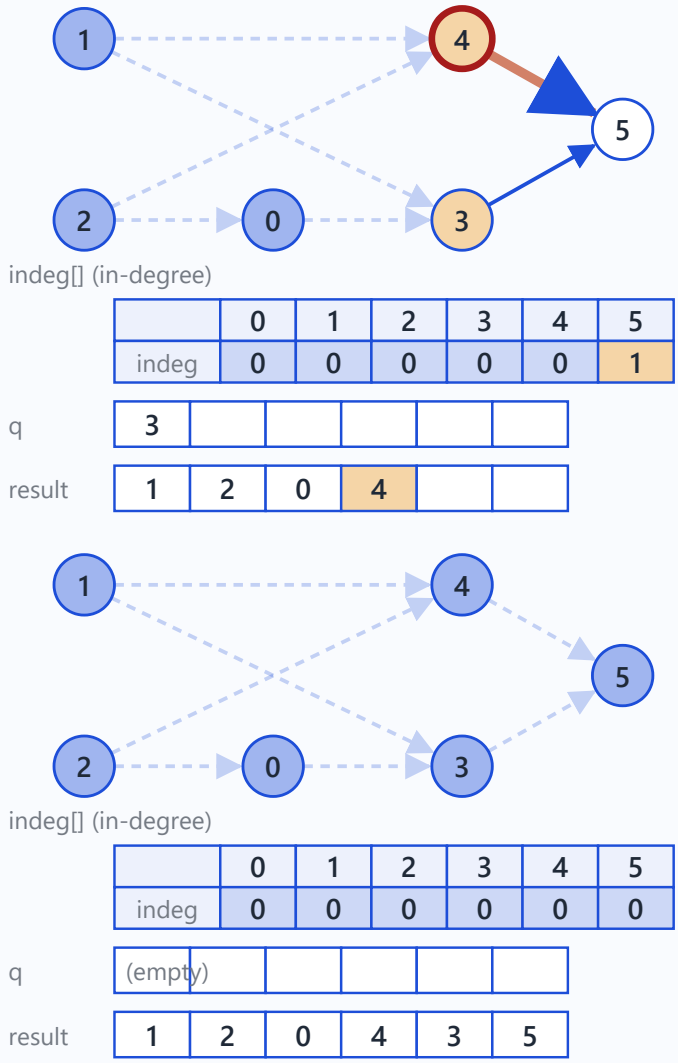


Figure 14.26: Four moments of `kahn()`. (a) 2 has been dequeued and added to the result; its arrows are not yet processed, and the queue is empty. (b) 2 → 4 is deleted too; queue 0, 4. (c) 4 → 5 is deleted; 5 still waits for 3. (d) The end: result 1 2 0 4 3 5.

The result `1 2 0 4 3 5` has size $6 = n$: no cycle, and every arrow's source comes first (for $2 \to 0$, 2 is second, 0 third). One enqueue per vertex, one decrement per arrow: $O(n + m)$.

Example 14.8

With arrows $0 \to 1$, $1 \to 2$, $2 \to 0$, $2 \to 3$, the in-degrees are `[1, 1, 1, 1]`; the queue starts empty and the loop never runs. The empty result, $0 < 4$, reveals a cycle. Vertex 3 is off the cycle but depends on 2, so it cannot be ordered either.

DFS also sorts topologically: a vertex's call finishes after those of everything reachable from it, so reverse finishing order is a topological order, with cycles caught by the back-edge check of the next section.

Common mistake

Not checking Kahn's result. On a cyclic graph it returns a short list without error; unless its size is compared with n, the list passes for an order and the cycle's courses silently vanish from the plan.

14.3.14 Detecting cycles in a directed graph

In an undirected graph, DFS meeting a marked neighbor other than its parent means a cycle. In a directed graph an arrow to a marked vertex is not enough: the vertex must lie on the current DFS path. Three colors tell the cases apart:

- **white (0):** not yet visited;
- **gray (1):** visited and on the current DFS path; its call has not finished;
- **black (2):** visited and finished; everything reachable from it has been examined.

```
// color: 0 white, 1 gray (on path), 2 black
boolean hasCycle(int v, int[] color) {
  color[v] = 1;
  for (int u : adj.get(v)) {
    if (color[u] == 1) return true;  // back edge
    if (color[u] == 0 && hasCycle(u, color))
      return true;
  }
  color[v] = 2;
  return false;
}
```

A gray head means the DFS path leads from it to the current vertex, and the arrow closes a cycle: a **back edge**. A black head has no path back, or the current vertex would have been visited during its call, so it closes no cycle.



Figure 14.27: Arrows 0 → 1, 1 → 2, 0 → 3, 3 → 2, 3 → 4, 4 → 0. When 4 → 0 is examined, 0 is gray, on the current path, so the cycle 0 → 3 → 4 → 0 is found. 1 and 2 are black.

Step	Event	`color` (0…4)
1	0 turns gray	[1, 0, 0, 0, 0]
2	0 → 1: 1 white, call; 1 turns gray	[1, 1, 0, 0, 0]
3	1 → 2: 2 white, call; 2 turns gray	[1, 1, 1, 0, 0]
4	2 has no arrows: 2 black, `false`	[1, 1, 2, 0, 0]
5	1's arrows done: 1 black, `false`	[1, 2, 2, 0, 0]
6	0 → 3: 3 white, call; 3 turns gray	[1, 2, 2, 1, 0]
7	3 → 2: 2 black, no cycle, continue	[1, 2, 2, 1, 0]
8	3 → 4: 4 white, call; 4 turns gray	[1, 2, 2, 1, 1]
9	4 → 0: 0 **gray** → back edge, `true`	[1, 2, 2, 1, 1]
10	`true` is passed back to 3, then to 0	[1, 2, 2, 1, 1]

Table 14.16: Trace of `hasCycle(0, color)`.

Step 7 shows the subtlety: 3 → 2 leads to a visited vertex, and asking only "visited?" would report a false cycle, but 2 is black, with no path back to 3. In step 9, 0 is gray: 0 → 3 → 4 is still on the call stack, and 4 → 0 closes the cycle 0 → 3 → 4 → 0. `true` returns at once through every level.

Common mistake

Forgetting `color[v] = 2`, so finished vertices stay gray. In the DAG 0 → 1, 0 → 2, 1 → 2, vertices 0, 1, 2 turn gray, and 2 and 1 return still gray; back at 0, the arrow 0 → 2 finds `color[2] == 1` and reports a nonexistent cycle. Directed graphs need gray and black, not just visited and unvisited.

The cost is that of DFS: each vertex turns gray and black once, each arrow is examined once. If some vertices are unreachable from 0, `hasCycle` restarts at each white one, still $O(n + m)$.

14.4 Shortest paths

A map app's route, a router's fastest packet path, a game character's way to its goal: one graph problem. Without weights, length is the edge count and BFS solves it; with weights, length is the weight sum and Dijkstra's algorithm solves it.

14.4.1 The unweighted shortest-path problem

Definition 14.12

The **single-source shortest-path** problem gives a source vertex s and asks for the length of the shortest path from s to every vertex, called that vertex's **distance** from s. A vertex unreachable from s has distance ∞.

Without weights, each edge counts 1, and BFS solves the problem: a vertex on level k has distance exactly k.

Theorem 14.2 (BFS and shortest paths)

In an unweighted graph, BFS from s discovers the vertices in order of their distance from s. When a vertex is first reached from a vertex w, its distance is *distance(w) + 1*.

Proof. The queue always holds distance-k vertices in front and distance-$k + 1$ vertices behind. Initially it holds only s, at 0; when w at distance k leaves, its undiscovered neighbors join the back with $k + 1$, keeping the pattern. Take u at distance $k + 1$: a path of $k + 1$ edges reaches it, its next-to-last vertex w' at distance k. No vertex closer than k is adjacent to u, or u would be closer, and all distance-k vertices leave before any at $k + 1$. So u is discovered from distance k and gets $k + 1$. □

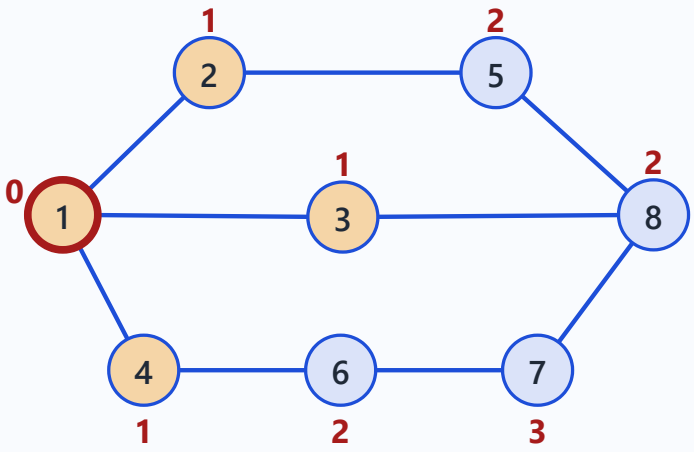


Figure 14.28: An unweighted graph with eight vertices and source 1. The edges are 1–2, 1–3, 1–4, 2–5, 3–8, 4–6, 5–8, 6–7, 7–8. The number beside each vertex is its distance from 1: 1 for 2, 3, 4; 2 for 5, 6, 8; 3 for 7.

Two three-edge paths reach 7: 1 → 4 → 6 → 7 and 1 → 3 → 8 → 7. BFS finds one, depending on list order, but the length is 3 either way: shortest paths need not be unique, shortest lengths are.

14.4.2 Algorithm and code

```
dist[every v] = ∞
dist[s] = 0
queue.add(s)
while queue is not empty:
  w = queue.remove()
  for each neighbor u of w:
    if dist[u] == ∞:
      dist[u] = dist[w] + 1
      queue.add(u)
return dist
```

This is BFS with a distance array as the visited array, "distance = ∞" meaning "unvisited." A newly discovered vertex gets its discoverer's distance plus one, which by Theorem 14.2 is final.

```
Map<Integer, Integer> shortestPath(int s) {
  final int INF = Integer.MAX_VALUE;  // ∞
  Map<Integer, Integer> dist =
      new HashMap<>();
  for (int v : graph.keySet())
    dist.put(v, INF);
  dist.put(s, 0);
  Queue<Integer> q = new ArrayDeque<>();
  q.offer(s);
  while (!q.isEmpty()) {
    int w = q.poll();
    for (int u : graph.get(w))
      if (dist.get(u) == INF) {
        dist.put(u, dist.get(w) + 1);
        q.offer(u);
      }
  }
  return dist;
}
```

Here the graph is a `Map<Integer, List<Integer>>` named `graph`, from vertex to neighbor list; vertices start at 1, hence a map rather than an array.

- `INF` stands for ∞ as `Integer.MAX_VALUE`.
- Every distance starts at `INF`; the source gets 0 and is enqueued (`offer` appends, like `add`).
- The main loop is BFS: each neighbor still at `INF` gets `dist.get(w) + 1` and is enqueued.

The cost is BFS's, *O(n + m)*, with `HashMap` operations *O(1)* on average.

Common mistake

Adding 1 to ∞. `Integer.MAX_VALUE + 1` overflows to −2,147,483,648. In the code, the `w` that gets 1 added has left the queue, so its distance is finite. Weighted graphs are riskier, which is why Dijkstra's Java version uses `long` distances and relaxes only from settled vertices.

14.4.3 Code trace: shortestPath(1)

Let the source be 1 in the graph of Figure 14.28, with neighbor lists in increasing order: 1: [2, 3, 4] · 2: [1, 5] · 3: [1, 8] · 4: [1, 6] · 5: [2, 8] · 6: [4, 7] · 7: [6, 8] · 8: [3, 5, 7].

Removed `w`	Neighbors	Updated	dist (1…8)	Queue
—	—	dist[1] = 0	0 ∞ ∞ ∞ ∞ ∞ ∞ ∞	[1]
1	2, 3, 4	2, 3, 4 ← 0 + 1 = 1	0 1 1 1 ∞ ∞ ∞ ∞	[2, 3, 4]
2	1, 5	5 ← 1 + 1 = 2	0 1 1 1 2 ∞ ∞ ∞	[3, 4, 5]
3	1, 8	8 ← 1 + 1 = 2	0 1 1 1 2 ∞ ∞ 2	[4, 5, 8]
4	1, 6	6 ← 1 + 1 = 2	0 1 1 1 2 2 ∞ 2	[5, 8, 6]
5	2, 8	— (both known)	0 1 1 1 2 2 ∞ 2	[8, 6]
8	3, 5, 7	7 ← 2 + 1 = 3	0 1 1 1 2 2 3 2	[6, 7]
6	4, 7	— (both known)	0 1 1 1 2 2 3 2	[7]
7	6, 8	—	0 1 1 1 2 2 3 2	[]

Table 14.17: Trace of `shortestPath(1)`. The distance column lists the distances of vertices 1…8 in order (∞ = `INF`).

Three points stand out. When 2 leaves, neighbor 1 keeps distance 0: in BFS the first distance found is shortest. When 5 leaves, neighbor 8 already has 2; via 5 it would be *2 + 1 = 3*, and the code does not even look, since 8 is no longer `INF`.

And 8, enqueued before 6, writes 7's distance, so the path found is 1 → 3 → 8 → 7; another list order might give 1 → 4 → 6 → 7, still 3. The final row matches Figure 14.28.

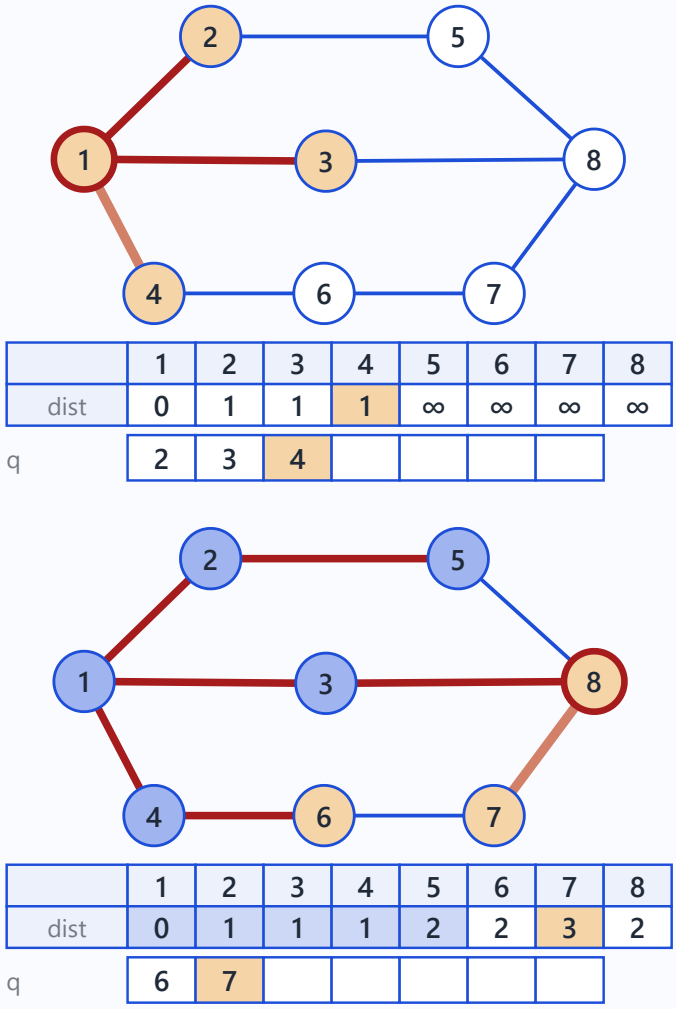


Figure 14.29: Two moments of `shortestPath(1)`. (a) 1 processed; 2, 3, 4 have distance 1, queue 2, 3, 4. (b) 8 processed; 7 gets distance 3, queue 6, 7.

The code returns distances, not paths; for paths, store each vertex's discoverer in a `prev` array. Dijkstra does the same, and 14.4.8 traces rebuilding a path backward from the target.

14.4.4 The weighted shortest-path problem

With weights the problem is the same, but length is the sum of edge weights: distances, times, or costs.

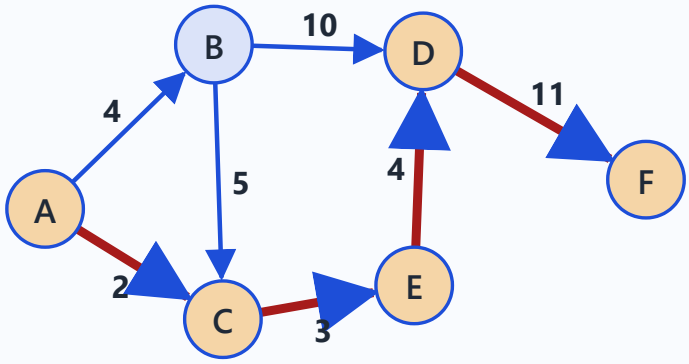


Figure 14.30: A weighted directed graph with arrows A → B: 4, A → C: 2, B → C: 5, B → D: 10, C → E: 3, E → D: 4, D → F: 11. The shortest path from A to F is A → C → E → D → F: *2 + 3 + 4 + 11 = 20*.

Example 14.9

In Figure 14.30 BFS finds A → B → D → F, the fewest edges (three), but it weighs *4 + 10 + 11 = 25*, while the four-edge A → C → E → D → F weighs 20. Fewest edges is not least weight; BFS is not enough.

The solution is the algorithm Edsger W. Dijkstra devised in 1956 and published in 1959. It has one requirement: **edge weights must not be negative**, for reasons that emerge in its correctness proof (14.4.9).

14.4.5 Dijkstra's algorithm: idea and pseudocode

Dijkstra's algorithm keeps a **tentative distance** for each vertex, the length of the shortest path found so far: 0 for the source, ∞ for the rest. Each step:

1. picks, among vertices not yet **settled**, the one with the smallest tentative distance, whose distance is now final;
2. updates its neighbors' distances by **relaxation**.

Definition 14.13

To **relax** an edge *(u, v)* is to test whether going through *u* beats the known path to *v*: compute *alt = dist[u] + weight(u, v)*, and if *alt < dist[v]*, set *dist[v] = alt* and *prev[v] = u*.

The name pictures an upper bound loosening like a spring. A tentative distance never drops below the true distance; each relaxation either leaves it alone or moves it closer.

```
Dijkstra(G, s):
  for each vertex v:
    dist[v] = ∞
    prev[v] = undefined
  dist[s] = 0
  Q = all vertices  // priority queue
  while Q is not empty:
    u = Q.removeMin()   // smallest dist[u]
    for each neighbor v of u in Q:
      alt = dist[u] + weight(u, v)
      if alt < dist[v]:
        dist[v] = alt
        prev[v] = u
        // decrease v's priority in Q
```

`Q` is a priority queue of unsettled vertices keyed by tentative distance; `Q.removeMin()` returns the closest. When relaxation lowers a distance, the vertex's priority must drop too. `prev[]` records each vertex's predecessor on its shortest path, and paths are read backward from it.

Common mistake

Prioritizing by vertex number. The queue must return the vertex with the smallest distance, not the smallest number; that is why the comparator of the Java `PriorityQueue<long[]>` compares the entry's second element, the distance.

14.4.6 Dijkstra in Java

```
long[] dijkstra(List<List<int[]>> adj, int s, int[] prev) { // weights ≥ 0
    long[] d = new long[adj.size()];
    Arrays.fill(d, Long.MAX_VALUE);  d[s] = 0;  Arrays.fill(prev, -1);
    PriorityQueue<long[]> pq =                    // {vertex, distance}
        new PriorityQueue<>((a, b) -> Long.compare(a[1], b[1]));
    pq.add(new long[]{s, 0});
    while (!pq.isEmpty()) {
        long[] e = pq.poll();  int u = (int) e[0];
        if (e[1] > d[u]) continue;                  // stale entry: skip
        for (int[] k : adj.get(u)) {
            int v = k[0];  long alt = d[u] + k[1];
            if (alt < d[v]) { d[v] = alt; prev[v] = u; pq.add(new long[]{v, alt}); }
        }
    }
    return d;
}
```

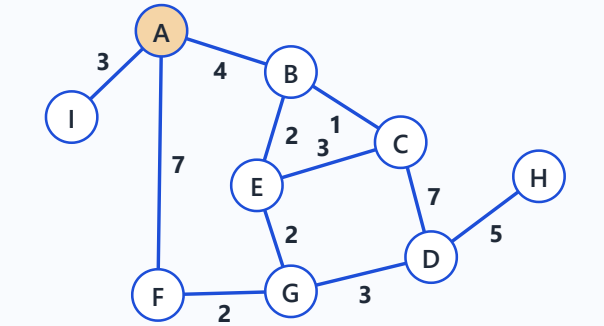
- **Input.** `adj.get(u)` lists *u*'s neighbors as `int[]{v, w}` pairs, neighbor and weight. The caller allocates `prev`; `dijkstra` fills it.
- **Setup.** The `long` array `d` is filled with `Long.MAX_VALUE`, standing for ∞; `d[s] = 0`; `prev` is filled with −1, "no predecessor."
- **Queue.** Entries are `{vertex, distance}` pairs; the comparator `Long.compare(a[1], b[1])` orders by distance, so `poll` returns the closest.
- **Stale check.** `if (e[1] > d[u]) continue;` skips an entry whose distance exceeds the vertex's current one.
- **Relaxation.** For each neighbor, `alt = d[u] + k[1]`; if `alt < d[v]`, the distance and `prev` are updated and a new `{v, alt}` entry is added.

Java's `PriorityQueue` has no decrease-key, so the code adds a new entry for *v* and leaves the old one. When the old entry surfaces, its distance exceeds the since-lowered `d[v]`: a **stale entry**, skipped by `continue`. This is lazy deletion: ignore old entries when they surface instead of removing them.

> **Common mistake**
>
> Taking the stale-entry line for a correctness requirement, or deleting it as useless. With non-negative weights, distances come out right without it: by then `d[u]` is smaller and the neighbors were relaxed with it, so `alt < d[v]` fails. The line saves rescanning stale entries' neighbors, which costs real time on dense graphs.

14.4.7 Code trace: dijkstra(adj, A)

Give the nine-vertex traversal graph weights: A–B: 4, A–F: 7, A–I: 3, B–C: 1, B–E: 2, C–D: 7, C–E: 3, D–G: 3, D–H: 5, E–G: 2, F–G: 2. The graph is undirected, each edge in both lists, and the source is A.



	A	B	C	D	E	F	G	H	I
d[]	0	∞	∞	∞	∞	∞	∞	∞	∞
prev	–	–	–	–	–	–	–	–	–

pq (shown by distance; faded = stale)

A,0

Figure 14.31: The start of the Dijkstra trace: `d[A] = 0`, all other distances ∞, `prev` empty, and only `{A, 0}` in the queue.

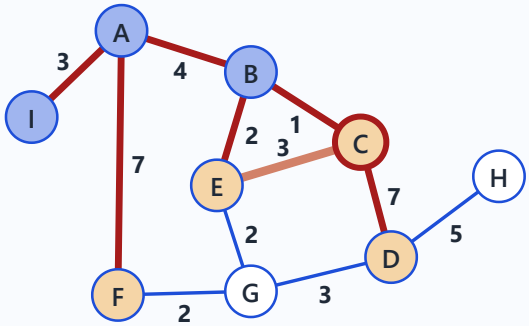
Each row below is one `poll`. The queue column shows the entries left after that row's relaxations, sorted by distance.

Removed entry	Status	Relaxations	Queue (after)
{A, 0}	A settled (0)	B: 0 + 4 = 4 < ∞ ✓; F: 0 + 7 = 7 ✓; I: 0 + 3 = 3 ✓	{I,3} {B,4} {F,7}
{I, 3}	I settled (3)	A: 6 ≥ 0	{B,4} {F,7}
{B, 4}	B settled (4)	A: 8 ≥ 0; C: 4 + 1 = 5 ✓; E: 4 + 2 = 6 ✓	{C,5} {E,6} {F,7}
{C, 5}	C settled (5)	B: 6 ≥ 4; D: 5 + 7 = 12 ✓; E: 8 ≥ 6	{E,6} {F,7} {D,12}
{E, 6}	E settled (6)	B: 8 ≥ 4; C: 9 ≥ 5; G: 6 + 2 = 8 ✓	{F,7} {G,8} {D,12}
{F, 7}	F settled (7)	A: 14 ≥ 0; G: 9 ≥ 8	{G,8} {D,12}
{G, 8}	G settled (8)	D: 8 + 3 = 11 < 12 ✓; E: 10 ≥ 6; F: 10 ≥ 7	{D,11} {D,12}
{D, 11}	D settled (11)	C: 18 ≥ 5; G: 14 ≥ 8; H: 11 + 5 = 16 ✓	{D,12} {H,16}
{D, 12}	**stale**: 12 > d[D] = 11	skipped	{H,16}
{H, 16}	H settled (16)	D: 21 ≥ 11	(empty)

Table 14.18: Trace of `dijkstra(adj, A)`.

The critical row is G's. D stands at 12, via C (A → B → C → D: *4 + 1 + 7*). Settling G relaxes D–G to *8 + 3 = 11 < 12*: D drops to 11, `prev[D] = G`, and `{D, 11}` joins the queue beside the old `{D, 12}`. `{D, 11}` surfaces first and settles D; `{D, 12}` follows, stale since *12 > 11*, and is skipped.

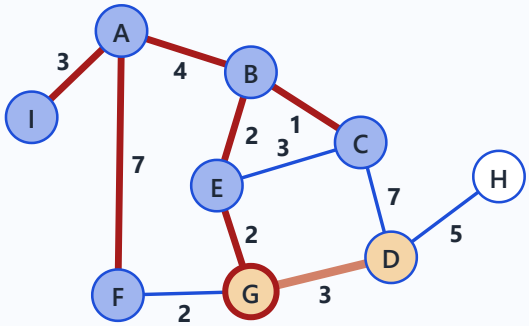
Vertices settle in distance order, A (0), I (3), B (4), C (5), E (6), F (7), G (8), D (11), H (16), the property behind Dijkstra's correctness. There were 10 `poll`s, 10 insertions (1 initial, 9 successful relaxations), and 22 relaxation attempts, *2m = 22*.



	A	B	C	D	E	F	G	H	I
d[]	0	4	5	12	6	7	∞	∞	3
prev	–	A	B	C	B	A	–	–	A

pq (shown by distance; faded = stale)

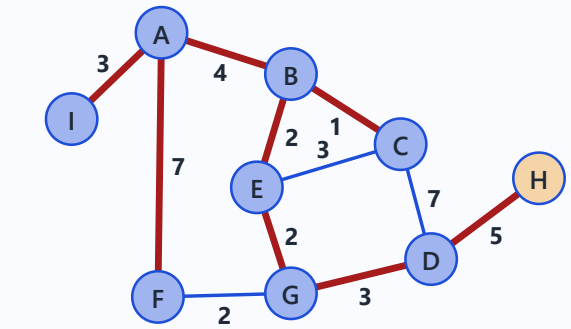
E,6 F,7 D,12



	A	B	C	D	E	F	G	H	I
d[]	0	4	5	11	6	7	8	∞	3
prev	–	A	B	G	B	A	E	–	A

pq (shown by distance; faded = stale)

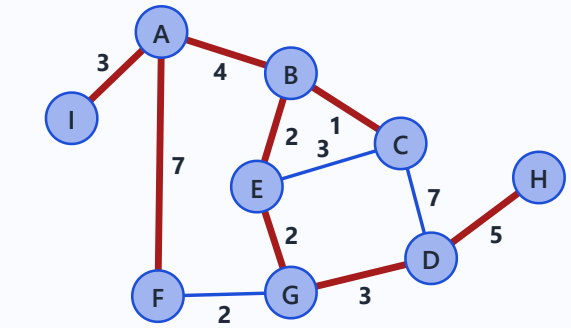
D,11 ~~D,12~~



	A	B	C	D	E	F	G	H	I
d[]	0	4	5	11	6	7	8	16	3
prev	–	A	B	G	B	A	E	D	A

pq (shown by distance; faded = stale)

H,16



	A	B	C	D	E	F	G	H	I
d[]	0	4	5	11	6	7	8	16	3
prev	–	A	B	G	B	A	E	D	A

pq (shown by distance; faded = stale)

(empty)

Figure 14.32: Four moments of the Dijkstra trace. (a) C is settled; `d[D] = 12` (through C). (b) A shorter path through G is found: `d[D] = 11`, `prev[D] = G`; the old `{D, 12}` entry is stale. (c) The stale entry comes out and is skipped. (d) Final state: the bold edges form the shortest-path tree.

	A	B	C	D	E	F	G	H	I
`d`	0	4	5	11	6	7	8	16	3
`prev`	–	A	B	G	B	A	E	D	A

Table 14.19: Dijkstra's result (source A).

`prev` holds the shortest-path tree: from H, H ← D ← G ← E ← B ← A, so A → B → E → G → D → H, five edges of length *4 + 2 + 2 + 3 + 5 = 16*. The four-edge A → F → G → D → H has length *7 + 2 + 3 + 5 = 17*.

14.4.8 Reconstructing the path

To get the path from *s* to *t*, follow `prev` from *t* until −1. The chain runs backward, so each vertex goes to the *front* of the list.

```
List<Integer> path(int[] prev, int s, int t) {
  LinkedList<Integer> p = new LinkedList<>();
  for (int v = t; v != -1; v = prev[v])
    p.addFirst(v);
  if (p.getFirst() != s) return List.of();
  return p;                  // s ... t
}
```

Starting at `v = t`, each round prepends `v` and steps back with `v = prev[v]`, stopping after the source, whose `prev` is −1. `LinkedList.addFirst` is constant time, so the cost is *O(path length)*, at most *O(n)*.

The final check catches unreachable targets: `prev[t] = -1` stops the chain at once, leaving *t* first, and a first item other than *s* yields an empty list.

Example 14.10

Run from A on the graph of Figure 14.30, Dijkstra leaves `prev` as B ← A, C ← A, D ← E, E ← C, F ← D. The call `path(prev, A, F)` goes:

1. `v = F` : list [F]; `v = prev[F] = D` .
2. `v = D` : list [D, F]; `v = prev[D] = E` .
3. `v = E` : list [E, D, F]; `v = prev[E] = C` .
4. `v = C` : list [C, E, D, F]; `v = prev[C] = A` .
5. `v = A` : list [A, C, E, D, F]; `v = prev[A] = -1` , and the loop ends.

Five rounds, five vertices. The first is A, so F is reachable: A → C → E → D → F, of length `d[F] = 2 + 3 + 4 + 11 = 20` .

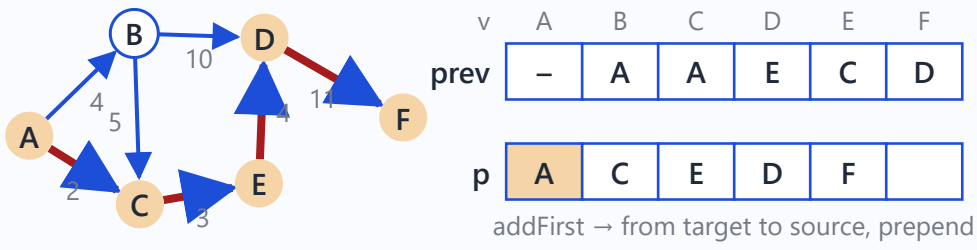


Figure 14.33: The end of path reconstruction. The `prev` chain has been followed from F to A, each vertex added to the front of the list: p = [A, C, E, D, F].

Common mistake

Reusing `prev` for another source. It is the shortest-path tree of the source Dijkstra ran from: `prev = [-1, 0, 0, 1, 3]`, built from 0, gives 0 → 1 → 3 → 4. Asked for 2 to 4, the chain 4, 3, 1, 0 never meets 2, and the result is empty; run Dijkstra from 2 instead.

14.4.9 Dijkstra: analysis and correctness

Time. Two implementations are common:

Implementation	Complexity	Why?
Priority queue (binary heap)	O((n + m) log n)	n removals of the minimum, at most m insertions or updates, each O(log n)
No queue (array scan)	$O(n^2 + m) = O(n^2)$	Each step scans n vertices for the closest; neighbor checks total m

Table 14.20: Two implementations of Dijkstra's algorithm.

With a heap, each successful relaxation adds an entry, at most $m + 1$ in all, each insertion or removal $O(\log m)$. As $m \le n^2$, $\log m \le 2 \log n$, giving $O((n + m) \log n)$.

Sparse graphs ($m \approx n$) favor the heap, $O(n \log n)$ against the array scan's $O(n^2)$; on very dense ones ($m \approx n^2$) the heap rises to $O(n^2 \log n)$ and the $O(n^2)$ scan can win.

Theorem 14.3 (Correctness of Dijkstra's algorithm)

If no edge weight is negative, then when Dijkstra's algorithm settles a vertex, its tentative distance equals its true distance from the source.

Proof. Let u be the unsettled vertex of smallest tentative distance, and take any path from s to u. It leaves the settled region somewhere first; let y be the first vertex outside and x its predecessor. (x, y) was relaxed when x settled, so $dist[y]$ is at most the length of the path up to y, and $dist[u] \le dist[y]$ by the choice of u. The rest of the path has non-negative edges, length ≥ 0. So the path is at least $dist[y] \ge dist[u]$ long: none is shorter than $dist[u]$. □

Non-negativity is used once: "length ≥ 0." A negative edge could later shorten a settled vertex's distance, and the pseudocode then errs. The stale-entry Java version may recover by reprocessing vertices, but loses its time bound, and a negative cycle makes it run forever. Negative weights call for the Bellman-Ford algorithm.

Example 14.11

Take arrows s → a: 2, s → b: 3, b → a: −2, a → c: 1. Dijkstra settles s (0), then a (2), giving c distance 3, then b (3). Via b → a, *3 + (−2) = 1* is a's true distance, but a is already settled at 2, and the pseudocode never revisits it, so c stays at 3 instead of *1 + 1 = 2*. Here the Java version re-adds a with 1 and processes it again, correcting c to 2: right answer, but a vertex processed twice.

14.5 Minimum spanning trees

A municipality wants water in every village; pipe between any two villages has a cost, and water need only reach each village by some route. Which network costs least? This minimum spanning tree problem also arises in power and communication networks, clustering, and inside other algorithms.

14.5.1 Spanning trees and the MST

Definition 14.14

A **spanning tree** of a connected graph is a connected, acyclic subgraph containing all its vertices; with n vertices it has exactly $n - 1$ edges. A **minimum spanning tree** (MST) of a weighted graph is a spanning tree of smallest total edge weight.

Why no cycles? With positive weights, dropping a cycle edge lowers the total while the rest of the cycle keeps its endpoints connected, so the cheapest connected subgraph is a tree. The $n - 1$ is the lower bound of 14.1.7, met exactly by acyclic connected graphs.

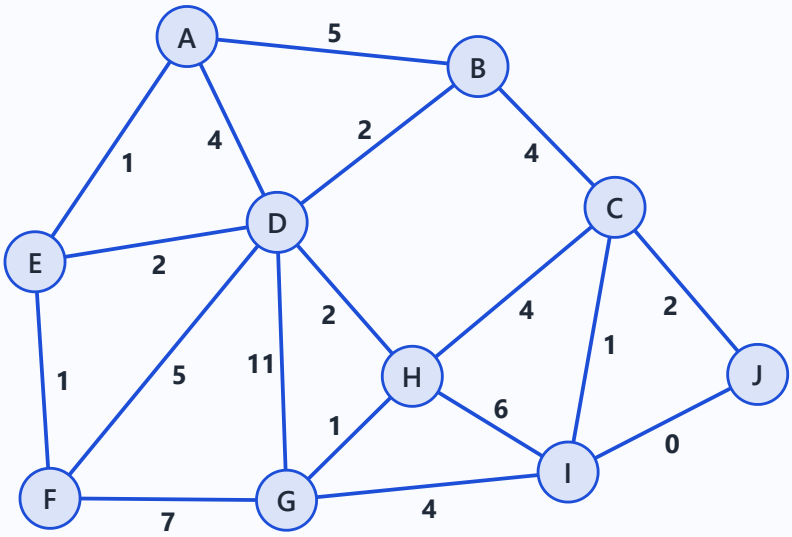


Figure 14.34: An undirected graph with ten vertices and eighteen weighted edges. By weight: I–J 0; A–E, C–I, E–F, G–H 1; B–D, C–J, D–E, D–H 2; A–D, B–C, C–H, G–I 4; A–B, D–F 5; H–I 6; F–G 7; D–G 11.

The edges weigh 62 in all; the MST takes nine of them, weighing 14. Spanning trees are far too many to try, so the classic algorithms, Kruskal's and Prim's, are greedy: each step takes the best-looking choice and never revisits it. Greed works here thanks to this property.

> **Theorem 14.4 (Cut property)**
>
> Split the vertices of a graph into two nonempty groups. Among the edges joining the two groups, one of smallest weight belongs to some minimum spanning tree.

Proof. Let $e = (u, v)$ be the lightest crossing edge and T an MST without it. T has a path from u to v, which crosses between the groups along some edge f. Adding e and removing f leaves a spanning tree: the one cycle e creates is broken by removing f. As $weight(e) \le weight(f)$, the new tree weighs no more than T, so it is an MST containing e. □

Kruskal and Prim apply this theorem in two ways. In Prim, one group is the tree built so far; in Kruskal, the groups are the trees containing the two endpoints of the chosen edge.

14.5.2 Disjoint sets (Union-Find)

Kruskal tries edges from lightest to heaviest, asking of each whether it closes a cycle, that is, whether chosen edges already connect its endpoints. **Disjoint sets** (**union-find**) answer this quickly, managing non-overlapping sets with two operations:

- `find(x)` returns the **representative** of x's set; two items are in the same set exactly when `find` gives them the same representative.

- `union(u, v)` merges the sets of u and v and returns `true` if they differ; if they are already the same set, it does nothing and returns `false`.

The slides use only this behavior; the class below is an implementation that reproduces the parent arrows of their trace.

```
class UnionFind {
    private final int[] parent, size;
    UnionFind(int n) {                      // each vertex its own set
        parent = new int[n];  size = new int[n];
        for (int i = 0; i < n; i++) { parent[i] = i; size[i] = 1; }
    }
    int find(int x) {                       // find the root, compress the path
        if (parent[x] != x) parent[x] = find(parent[x]);
        return parent[x];
    }
    boolean union(int u, int v) {
        int a = find(u), b = find(v);
        if (a == b) return false;           // same set: would close a cycle
        if (size[a] > size[b]) { int t = a; a = b; b = t; }
        parent[a] = b;  size[b] += size[a];      // smaller tree under the larger
        return true;
    }
}
```

Each set is a tree: `parent[x]` is x's parent, and the root, its own parent, represents the set. Initially there are n one-vertex trees. Two refinements give speed:

- **Union by size.** The smaller tree's root goes under the larger's, keeping heights at most $\log n$; on a tie, u's root goes under v's.
- **Path compression.** `find` links every vertex on its way directly to the root, so repeat queries take one step.

Together they make each operation cost $O(\alpha(n))$ over a sequence, where α, the inverse Ackermann function, stays ≤ 4 even for n the number of atoms in the universe: in practice, constant time.

14.5.3 Kruskal's algorithm

```
Kruskal(G):
  T = ∅          // chosen edges
  for each vertex v: makeSet(v)
  sort the edges by weight
  for each (u, v), lightest first:
    if find(u) != find(v): // no cycle
      T.add((u, v))
      union(u, v)
    if |T| == n - 1: break
  return T  // MST (if disconnected: forest)
```

Kruskal's rule: **pick the lightest edge that closes no cycle.** Starting from n one-vertex sets, it tries edges by increasing weight; an edge joining different sets enters T and merges them, one inside a set is skipped. After $n - 1$ edges the tree is done.

Midway, the chosen edges form a **forest** whose trees merge into one. A disconnected graph never reaches $n - 1$; every edge is tried, giving a **minimum spanning forest**, one tree per component.

Phase	Cost
Sorting the edges	O(m log m)
2m `find` + (n − 1) `union`	O(m · α(n))
Total	**O(m log n)**

Table 14.21: Cost of Kruskal's algorithm.

Sorting dominates: as $\log m \le 2 \log n$, $O(m \log m) = O(m \log n)$, and the nearly linear Union-Find work changes nothing. A DFS per edge for cycle checks would cost $O(m \cdot n)$.

14.5.4 Kruskal step by step

We run Kruskal on the graph of Figure 14.34, trying equal-weight edges in the order of its caption.

Edge	Weight	Decision	Sets merged / reason	|T|	Total
I–J	0	accept	{I} + {J}	1	0
A–E	1	accept	{A} + {E}	2	1
C–I	1	accept	{C} + {I, J}	3	2
E–F	1	accept	{A, E} + {F}	4	3
G–H	1	accept	{G} + {H}	5	4
B–D	2	accept	{B} + {D}	6	6
C–J	2	**reject**	C and J already in {C, I, J}	6	6
D–E	2	accept	{B, D} + {A, E, F}	7	8

Edge	Weight	Decision	Sets merged / reason	|T|	Total
D–H	2	accept	{A, B, D, E, F} + {G, H}	8	10
A–D	4	**reject**	A and D in the same set	8	10
B–C	4	accept	large set + {C, I, J}	9	14

Table 14.22: The edges Kruskal tries, and its decisions.

C–J is rejected because C–I and I–J already connect C and J; it would close C–I–J–C. Likewise A–D, with A–E–D in place. Accepting B–C makes $|T| = 9 = n - 1$, total $0 + 1 + 1 + 1 + 1 + 2 + 2 + 2 + 4 = 14$, and the other seven edges (C–H, G–I, A–B, D–F, H–I, F–G, D–G) are never examined.

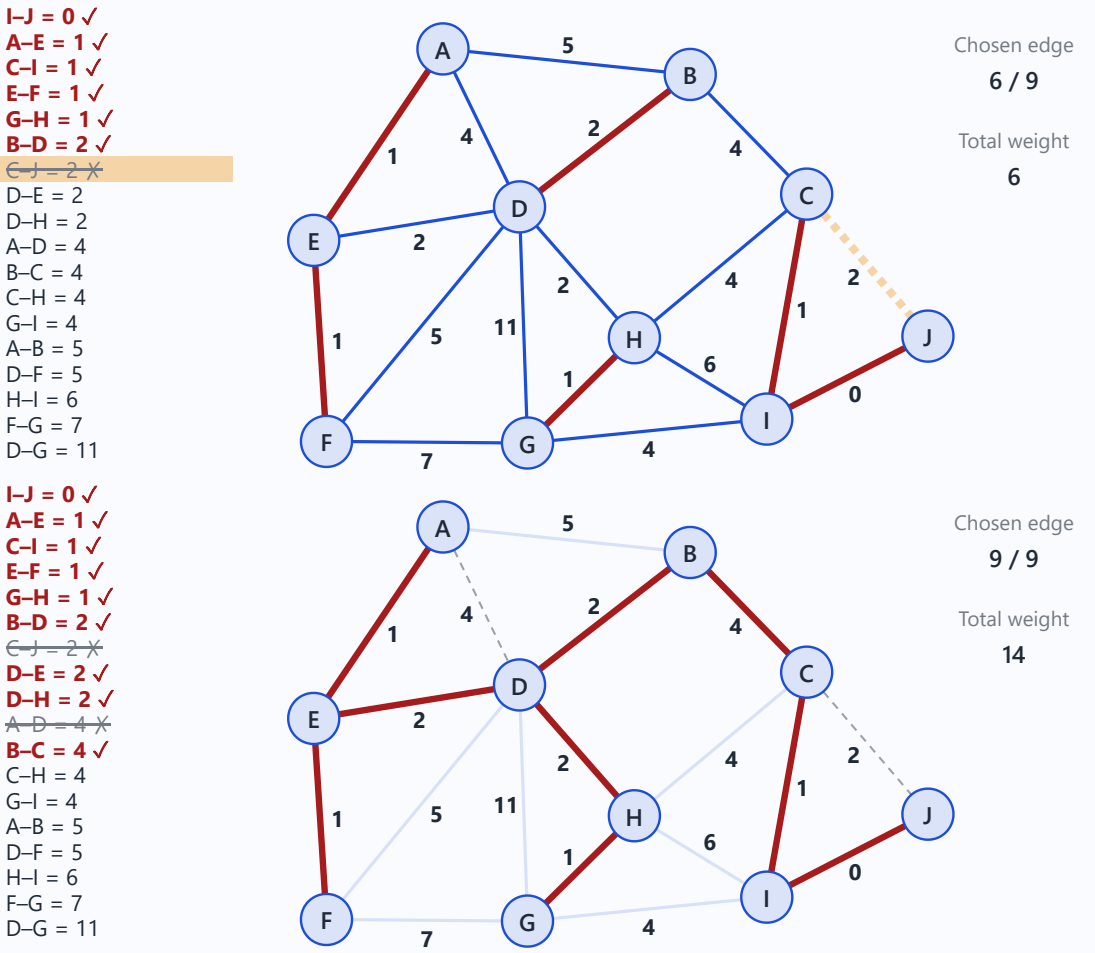


Figure 14.35: Two moments of Kruskal. Top: C–J is rejected, since C and J are in the same tree. Bottom: the final state, with the nine chosen edges A–E, E–F, D–E, B–D, D–H, G–H, B–C, C–I, I–J and total weight 14. In the list on the left, accepted edges are checked and rejected ones struck through.

Common mistake

Judging cycles by whether the endpoints already touch chosen edges. D and E both do (B–D, A–E), yet lie in different trees, so D–E is accepted. What counts is being in the same *set*.

14.5.5 Code trace: kruskal() and Union-Find

```
List<int[]> kruskal(int n, List<int[]> E) {
  E.sort((a, b) ->          // {u, v, w}
      Integer.compare(a[2], b[2]));
  UnionFind uf = new UnionFind(n);
  List<int[]> T = new ArrayList<>();
  for (int[] e : E) {
    if (uf.union(e[0], e[1]))  // no cycle
      T.add(e);
    if (T.size() == n - 1) break;
  }
  return T;
}
```

Edges are `{u, v, w}` arrays, sorted by weight; Java's object sort is **stable**, so ties keep input order. `uf.union(e[0], e[1])` checks and merges in one call, a `true` adding the edge; `break` ends the loop at $|T| = n - 1$.

Inside Union-Find, with A = 0, …, J = 9, the "linked" column shows which root `union` attaches to which.

Edge	`find` roots	`union`	Linked	Parents changed
I–J (0)	I, J	true	I → J (equal sizes)	I → J
A–E (1)	A, E	true	A → E (equal)	A → E
C–I (1)	C, J	true	C → J (1 < 2)	C → J
E–F (1)	E, F	true	F → E (1 < 2)	F → E
G–H (1)	G, H	true	G → H (equal)	G → H
B–D (2)	B, D	true	B → D (equal)	B → D
C–J (2)	J, J	**false**	—	—
D–E (2)	D, E	true	D → E (2 < 3)	D → E
D–H (2)	E, H	true	H → E (2 < 5)	H → E
A–D (4)	E, E	**false**	—	—
B–C (4)	E, J	true	J → E (3 < 7)	B → E (path compression), J → E

Table 14.23: The Union-Find forest during `kruskal()`.

For C–J, `find(C)` follows C → J and `find(J)` returns J: same root, `false`, skipped. In the last step `find(B)` follows B → D → E, path compression links B to E, and J's tree (3 vertices) goes under E's (7). The single root E is reached by I and C through J, by G through H.

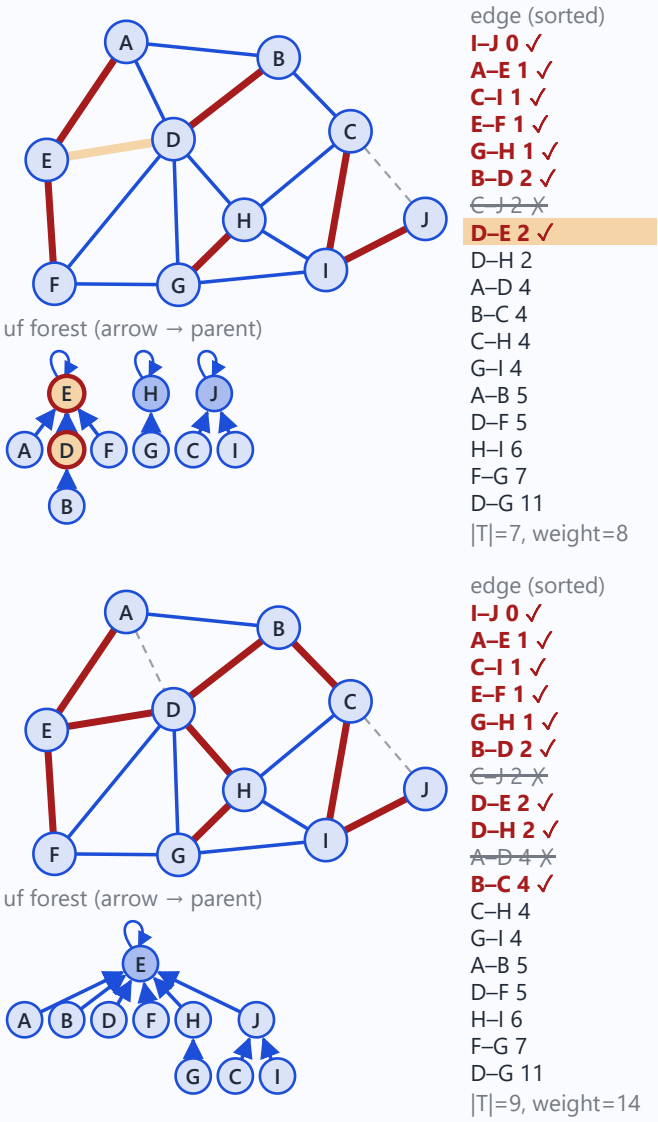


Figure 14.36: Two moments of the Union-Find forest (arrows point to parents). (a) D–E is accepted: D's tree goes under E. (b) Final state: one tree with root E. Path compression has pointed B's arrow directly at E.

Nine edges of weight 14 are returned; eleven of eighteen edges were tried, but sorting, the dominant cost, covers all eighteen.

14.5.6 A second example: airports

Goodrich, Tamassia, and Goldwasser give nine U.S. airports and 19 routes, weighted in miles; we want the shortest network connecting them all.

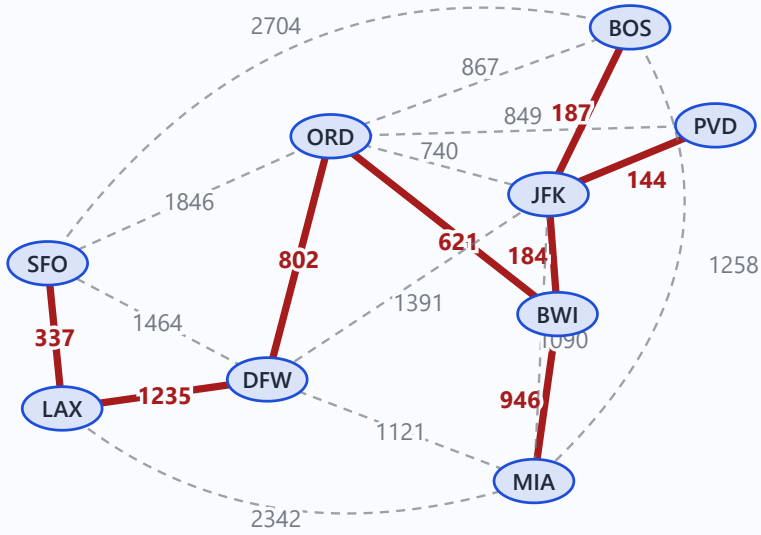


Figure 14.37: Nine airports and 19 routes. The bold edges are the MST found by Kruskal: JFK–PVD 144, JFK–BWI 184, JFK–BOS 187, SFO–LAX 337, ORD–BWI 621, ORD–DFW 802, BWI–MIA 946, LAX–DFW 1235. Dashed edges are not chosen.

Route (miles)	Decision	Reason
JFK–PVD 144, JFK–BWI 184, JFK–BOS 187	accept	four airports join around JFK
SFO–LAX 337	accept	a separate tree in the west
ORD–BWI 621	accept	ORD joins the eastern tree
ORD–JFK 740	reject	ORD and JFK already connected via BWI (cycle)
ORD–DFW 802	accept	DFW joins the eastern tree
ORD–PVD 849, ORD–BOS 867	reject	both ends in the eastern tree
BWI–MIA 946	accept	MIA joins the eastern tree
JFK–MIA 1090, DFW–MIA 1121	reject	both ends in the eastern tree
LAX–DFW 1235	accept	western and eastern trees merge; 8 = n − 1 edges

Table 14.24: Kruskal's decisions in the airport example.

The total is *144 + 184 + 187 + 337 + 621 + 802 + 946 + 1235 = 4456* miles. Thirteen routes are tried; the six longest (1258, 1391, 1464, 1846, 2342, 2704) never are. Short routes group nearby airports, and long ones enter only as bridges between groups: LAX–DFW, the shortest bridge between west and east, must be in the MST by the cut property.

14.5.7 Prim's algorithm

```
Prim(G, s):
  tree = {s}, T = ∅
  add s's edges to the queue
  while |T| < n − 1:
    (u, v) = lightest edge  // u in tree
    if v in tree: skip      // cycle
    add v to tree, (u, v) to T
    add v's edges leaving the tree
      to the queue
  return T
```

Prim grows one tree from a single vertex, adding each step the lightest edge to a vertex *outside* it. The cut property makes this safe: the tree must connect to the rest somehow, and this is the cheapest connection.

Like Dijkstra, Prim starts at a vertex and repeatedly takes a priority queue's smallest item, but the key differs: Dijkstra's is the *total* distance from the source, Prim's the weight of the *single edge* to the tree. Unlike Kruskal, there is always one tree, so “is the head in the tree?” detects cycles without Union-Find.

14.5.8 Prim in Java

The Java version in the slides is the **lazy** version, built on a priority queue.

```
List<int[]> prim(List<List<int[]>> adj, int s) {
  int n = adj.size();                 // adj.get(u): {v, w}
  boolean[] inTree = new boolean[n];
  PriorityQueue<int[]> pq = new PriorityQueue<>(
      (a, b) -> Integer.compare(a[2], b[2]));  // {u, v, w}
  List<int[]> T = new ArrayList<>();
  inTree[s] = true;
  for (int[] e : adj.get(s)) pq.add(new int[]{s, e[0], e[1]});
  while (!pq.isEmpty() && T.size() < n - 1) {
    int[] e = pq.poll();              // lightest candidate
    if (inTree[e[1]]) continue;       // stale: would close a cycle
    inTree[e[1]] = true;
    T.add(e);
    for (int[] f : adj.get(e[1]))
      if (!inTree[f[0]]) pq.add(new int[]{e[1], f[0], f[1]});
  }
  return T;
}
```

Entries are `{u, v, w}`: tree endpoint, candidate, weight (the comparator's only concern). `inTree` marks members.

1. The start `s` joins, and its edges are enqueued.
2. Until the queue empties or $n - 1$ edges are chosen, take the lightest entry.
3. If candidate `e[1]` is in the tree, the entry is **stale** (it would close a cycle), and `continue` skips it.
4. Otherwise the candidate joins, the edge enters T, and the new vertex's edges leaving the tree are enqueued.

"Lazy": when v joins, old entries to it stay queued and are skipped on removal, as with Dijkstra's stale entries.

Cost. Each edge is enqueued at most once, from the endpoint that joins first; later, `!inTree[f[0]]` blocks it. At most m insertions and removals at $O(\log m)$ give $O(m \log n)$ time and $O(m)$ queue memory. With a matrix and an array scan, Prim takes $O(n^2)$, better for very dense graphs.

> **Common mistake**
>
> Deleting the stale check (`if (inTree[e[1]]) continue;`). In Dijkstra it only saved time; in Prim it guarantees correctness. Without it, a leftover entry to v adds an edge between two tree vertices, creating a cycle.

14.5.9 Prim step by step

Start Prim at A on Figure 14.34, breaking ties alphabetically by tree endpoint, then candidate.

Step	Chosen edge	Joins tree	New candidates enqueued	Total
0	—	A	A–E 1, A–D 4, A–B 5	0
1	A–E (1)	E	E–D 2, E–F 1	1
2	E–F (1)	F	F–D 5, F–G 7	2
3	E–D (2)	D	D–B 2, D–G 11, D–H 2	4
4	D–B (2)	B	B–C 4	6
5	D–H (2)	H	H–C 4, H–G 1, H–I 6	8
6	H–G (1)	G	G–I 4	9
7	B–C (4) (stale A–D 4 skipped first)	C	C–I 1, C–J 2	13
8	C–I (1)	I	I–J 0	14
9	I–J (0)	J	—	14

Table 14.25: Prim from A (ties broken alphabetically).

When D joins in step 3, the entries A–D 4 and F–D 5 go stale but stay queued. In step 7, A–D leads the weight-4 entries alphabetically, is skipped as stale, and B–C is chosen. In step 9, J joins with $n - 1 = 9$ edges, leaving eight entries unexamined.

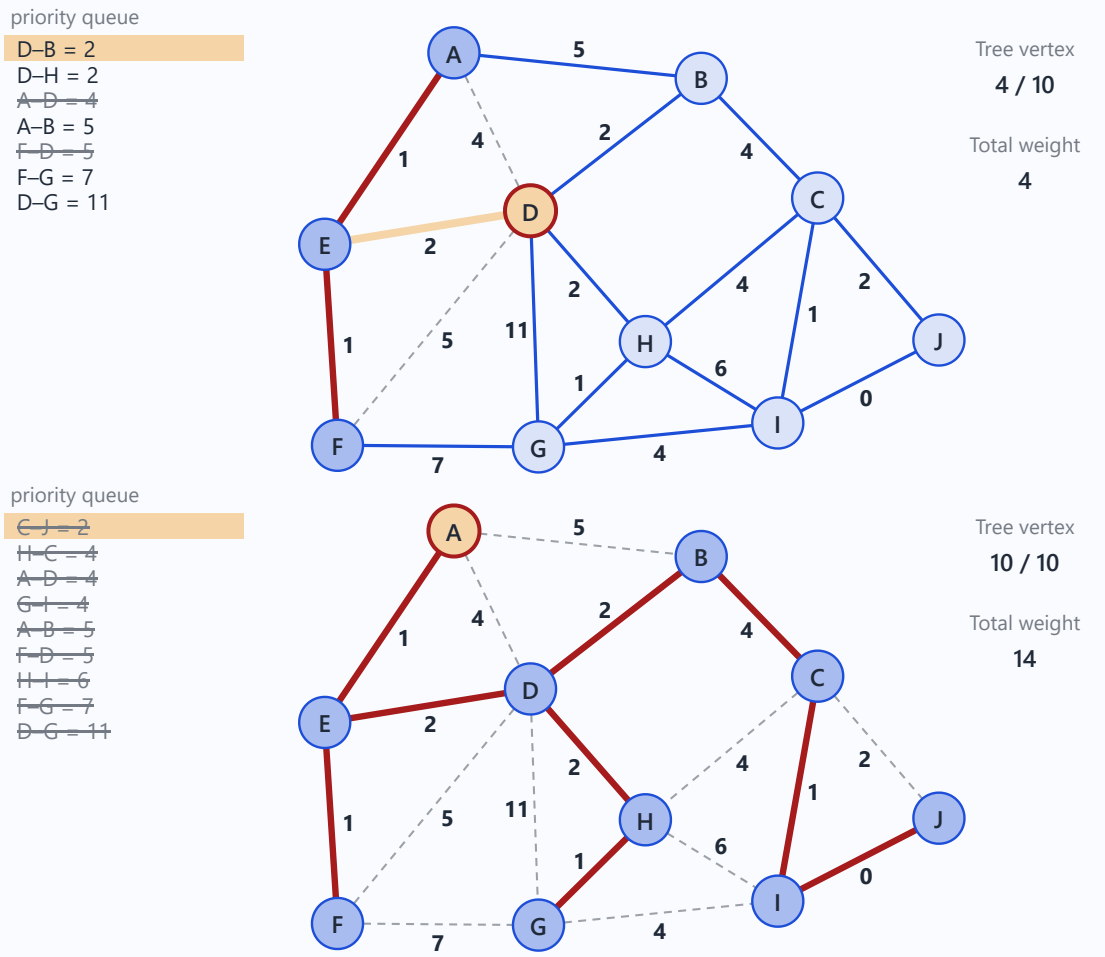


Figure 14.38: Two moments of Prim. Top: E–D is chosen and D joins the tree; struck-through entries have gone stale. Bottom: the final state, ten vertices in the tree, total weight 14.

Prim's tree, A–E, E–F, E–D, D–B, D–H, H–G, B–C, C–I, I–J, has Kruskal's edges in another order: Kruskal began with I–J (weight 0), which Prim reaches last.

Alphabetical ties are a hand-tracing rule. The Java code takes tied entries in `PriorityQueue`'s internal order, which guarantees nothing and depends on insertion order, hence on the lists. With alphabetical lists it picks G–I instead of B–C in step 7, then I–J and I–C: A–E, E–F, E–D, D–B, D–H, H–G, G–I, I–J, I–C. With edges listed in the order of the caption of Figure 14.34, it picks H–C in step 7. Each tree is an MST of weight 14: ties allow several MSTs, all of equal weight.

Example 14.12

Start Prim at J on the same graph (ties alphabetical).

1. J–I 0 and J–C 2 are enqueued; J–I brings in I, adding I–C 1, I–G 4, I–H 6.
2. I–C 1 brings in C, adding C–B 4 and C–H 4.
3. J–C 2 is now stale and skipped. Of C–B, C–H, and I–G at weight 4, C–B comes first, and B joins.

The first edges are J–I (0), I–C (1), C–B (4); then B–D, D–E, E–A, E–F, D–H, H–G, total 14. The start changes the building order, not the total.

14.5.10 Kruskal versus Prim

	Kruskal	Prim
Idea	tries edges lightest first, merging trees	grows one tree from a start vertex
Intermediate state	forest (several trees)	one tree
Data structures	sorted edge list and Union-Find	priority queue, mark array, adjacency list
Cycle check	`find(u) == find(v)`	is *v* already in the tree?
Cost	O(m log n) (sorting dominates)	O(m log n) with a heap; $O(n^2)$ with a matrix
When?	sparse graph, edges given as a list	adjacency list at hand; $O(n^2)$ version for very dense graphs

Table 14.26: Kruskal's and Prim's algorithms compared.

Both are greedy, rest on the cut property, and find the same total (14 here), though ties may yield different trees. The data's form also matters: edges read as a list suit Kruskal; adjacency lists with a natural start vertex suit Prim.

Common mistake

Confusing the MST with the shortest-path tree. Dijkstra's `prev` tree minimizes *path lengths* from the source; the MST minimizes the edge *sum*. In the triangle A–B 2, B–C 2, A–C 3, the MST is A–B, B–C (total 4), where A to C has length 4, though the edge A–C has length 3.

14.6 Summary

- A **graph** $G = (V, E)$ has n vertices and m edges. Unlike a tree, it has no root, may have cycles, and may offer several routes to a vertex. Edges may be directed or not, weighted or not.
- A vertex's **degree** counts its incident edges. By the handshake lemma $\Sigma\, deg(v) = 2m$; directed in- and out-degrees each sum to m. Simple undirected graphs have $m \le n(n-1)/2$, connected ones $m \ge n - 1$.

- A **path** is a sequence of vertices joined by edges; a **cycle** ends where it starts. A graph with paths between all vertex pairs is **connected**; a largest mutually reachable group is a **connected component**.
- An **adjacency matrix** takes $O(n^2)$ memory, with $O(1)$ edge queries and $O(n)$ neighbor visits; an **adjacency list** takes $O(n + m)$, with $O(\deg v)$ neighbor visits. Lists suit sparse graphs; matrices suit dense ones with frequent edge queries.
- **BFS** goes level by level with a queue, **DFS** deepest first with a stack or recursion, both $O(n + m)$ with lists. Vertices are marked on entering the queue or stack; without marks, cycles trap the traversal.
- Three applications follow. **Connected components**: a new traversal from each uncolored vertex. **Bipartiteness**: 2-coloring by BFS, possible exactly without odd cycles. **Directed cycle detection**: three-color DFS, where an arrow to a gray vertex is a back edge.
- **Topological orders** exist only for DAGs. **Kahn's algorithm** dequeues in-degree-0 vertices and deletes their arrows, in $O(n + m)$; fewer than n results reveal a cycle.
- BFS finds **unweighted shortest paths**, the first distance found being shortest. For non-negative weights, **Dijkstra** settles the closest unsettled vertex and relaxes its neighbors, in $O((n + m) \log n)$ with a heap or $O(n^2)$ by array scan; paths are read backward through `prev`.
- A **minimum spanning tree** connects all vertices with $n - 1$ edges of least total weight. **Kruskal** sorts edges and keeps those closing no cycle, via Union-Find; **Prim** grows one tree by its lightest outgoing edge. Both are $O(m \log n)$ and rest on the cut property.

Algorithm	Helper structure	Time (with adjacency lists)	Produces
BFS	queue	O(n + m)	level order, unweighted shortest paths
DFS	stack / recursion	O(n + m)	depth-first traversal, cycles, components
Topological sort (Kahn)	queue / DFS	O(n + m)	dependency order for a DAG
Dijkstra	priority queue	O((n + m) log n)	weighted shortest paths (weights ≥ 0)
Kruskal	sorting and Union-Find	O(m log n)	minimum spanning tree
Prim	priority queue	O(m log n)	minimum spanning tree

Table 14.27: The algorithms of this chapter.

The table shows the chapter's lesson: the data structure shapes the algorithm. One traversal skeleton becomes BFS with a queue, DFS with a stack, Dijkstra with a priority queue keyed by distance, and Prim with one keyed by edge weight.

14.7 Key terms

The glossary gives each term's Turkish equivalent and its use in other chapters.

graph | vertex, node | edge | adjacent | degree | in-degree | out-degree | directed graph (digraph) | weighted graph | path | cycle | connected component | complete graph | edge list | adjacency list | adjacency matrix | traversal | breadth-first search (BFS) | depth-first search (DFS) | bipartite graph | directed acyclic graph (DAG) | topological sort | relaxation | priority queue | spanning tree | minimum spanning tree (MST) | disjoint sets (union-find) | path compression | Kruskal's algorithm | Prim's algorithm

14.8 Test Yourself

1. What are the fewest and the most edges a simple, undirected, connected graph on 8 vertices can have?

   Answer

   At least $n - 1 = 7$ (the graph is then a tree), at most $n(n-1)/2 = 8 \cdot 7 / 2 = 28$ (the complete graph K_8).

2. How many cells are 1 in the adjacency matrix of the directed graph with arrows 0 → 1, 0 → 2, 2 → 1, 3 → 0? How do you read 0's out-degree and 1's in-degree from the matrix?

   Answer

   Each arrow fills one cell: 4 cells. 0's out-degree is the sum of row 0, `m[0][1] + m[0][2] = 2`; 1's in-degree is the sum of column 1, `m[0][1] + m[2][1] = 2`.

3. On the nine-vertex traversal graph (neighbors alphabetical), what does `bfs(E)` print? How many edges is A from E?

   Answer

   `E B C G A D F I H`. E's neighbors B, C, G form level 1; A (from B), D (from C), and F (from G) form level 2; I (from A) and H (from D) form level 3. A is on level 2, two edges from E (E → B → A).

4. On the same graph, what does recursive `dfs(F)` print?

Answer

`F A B C D G E H I`. From F the search descends to A, then A's first unvisited neighbor B, then C, D, G, and G's neighbor E. It backs up from E and G and visits H below D; finally, back at A, it visits A's last neighbor, I.

5. What order does `kahn()` return for the DAG with lists 0: [1, 2] · 1: [4] · 2: [4] · 3: [1] · 4: []?

Answer

The in-degrees are `[0, 2, 1, 0, 2]`, so the queue starts with 0, 3. Removing 0 gives `indeg[1] = 1` and `indeg[2] = 0`, enqueuing 2; removing 3 gives `indeg[1] = 0`, enqueuing 1; removing 2 gives `indeg[4] = 1`; removing 1 gives `indeg[4] = 0`, enqueuing 4. The result is `0 3 2 1 4`; its size is 5 = n, so there is no cycle.

6. On the graph with arrows 0 → 1, 1 → 2, 2 → 3, 3 → 1, while examining which arrow does `hasCycle(0, color)` return `true`? What is `color` at that moment?

Answer

0, 1, 2, 3 turn gray in turn: `color = [1, 1, 1, 1]`. When 3 → 1 is examined, 1 is gray, on the current path; the arrow is a back edge, and the cycle found is 1 → 2 → 3 → 1.

7. Run the Java Dijkstra from A on the undirected graph A–B 1, A–C 4, B–C 2, B–D 6, C–D 3. What are the final `d` and `prev`? How many stale entries leave the queue?

Answer

`d = [0, 1, 3, 6]` (A, B, C, D) and `prev = [-, A, B, C]`. Processing A sets C to 4 and B to 1; processing B lowers C to 3 (adding `{C, 3}`) and sets D to 7; processing C lowers D to 6 (adding `{D, 6}`). The old entries `{C, 4}` and `{D, 7}` come out stale and are skipped: 2 stale entries.

8. On the graph A–B 3, A–C 1, B–C 3, B–D 2, C–D 4, C–E 5, D–E 6, which edges does Kruskal choose, and what is the total weight? What changes if the tied edges A–B and B–C are tried in the other order?

Answer

In order: A–C 1 (accept), B–D 2 (accept), A–B 3 (accept, merging {A, C} and {B, D}), B–C 3 (reject), C–D 4 (reject), C–E 5 (accept). Four edges are chosen and D–E is never examined; the total is *1 + 2 + 3 + 5 = 11*. Trying B–C first accepts B–C and rejects A–B: a different tree (A–C, B–D, B–C, C–E), still totaling 11.

9. On the graph of the previous question, in what order does lazy Prim from A choose edges (ties alphabetical), and how many stale entries does it skip?

   Answer

   A–C 1, A–B 3, B–D 2, C–E 5. After A–B is chosen, C–B 3 remains queued and is stale, since B is in the tree; when D joins, C–D 4 goes stale too. Both come out before C–E and are skipped: 2 stale entries. The total is again 11.

10. In a graph whose edges all weigh 5, which algorithm can replace Dijkstra for single-source shortest paths, and why?

    Answer

    BFS. With equal weights a path's length is *5 × number of edges*, so the path with the fewest edges is also the shortest; multiply BFS's distances by 5. BFS runs in $O(n + m)$, saving the $\log n$ factor of heap-based Dijkstra.

14.9 Further reading

- M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, *Data Structures and Algorithms in Java* (6th ed., 2014), Chapter 14: the graph ADT, representations, DFS and BFS, topological sorting, Dijkstra, Kruskal, and Prim, with Java code; the airport example comes from this chapter.
- R. Sedgewick and K. Wayne, *Algorithms* (4th ed., 2011), Chapter 4: undirected and directed graphs, MSTs, and shortest paths in separate sections, with a comparison of the lazy and eager versions of Prim.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms* (4th ed., 2022), Chapters 20–22: formal correctness proofs for BFS and DFS, the cut property, and the Bellman-Ford algorithm; the analysis of disjoint sets is in Chapter 19.

14.10 References

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022.
2. E. W. Dijkstra, “A note on two problems in connexion with graphs,” *Numerische Mathematik*, 1, 269–271, 1959.
3. M. T. Goodrich, R. Tamassia, M. H. Goldwasser, *Data Structures and Algorithms in Java*, 6th ed., Wiley, 2014.
4. A. B. Kahn, “Topological sorting of large networks,” *Communications of the ACM*, 5(11), 558–562, 1962.
5. J. B. Kruskal, “On the shortest spanning subtree of a graph and the traveling salesman problem,” *Proceedings of the American Mathematical Society*, 7(1), 48–50, 1956.

6. R. C. Prim, “Shortest connection networks and some generalizations,” *Bell System Technical Journal*, 36(6), 1389–1401, 1957.

7. R. Sedgewick, K. Wayne, *Algorithms*, 4th ed., Addison-Wesley, 2011.

APPENDIX A

Glossary of Terms

The English terms used in the book, each with the Turkish equivalent used in the Turkish edition. The number on the right is the chapter where the term is explained in detail. A searchable version of the glossary is on the course page.

A

abstract data type (ADT) → *soyut veri tipi* 1

accumulator → *biriktirici* *(kuyruk özyinelemesinde ara sonucu taşıyan parametre)* 4

adaptable priority queue → *uyarlanabilir öncelikli kuyruk* 11

adjacency list → *komşuluk listesi* 14

adjacency matrix → *komşuluk matrisi* 14

adjacent → *bitişik* 14

aging → *yaşlandırma* 11

algorithm → *algoritma* 1

algorithm analysis → *algoritma analizi* 2

aliasing → *takma ad* 1

amortized analysis → *amortize analiz* 6

ancestor → *ata* 9

array → *dizi* 3

array covariance → *dizi kovaryansı* 3

ASCII (American Standard Code for Information Interchange) → *ASCII* 1

asymptotic → *asimptotik* 2

autoboxing → *otomatik kutulama* 6

average case → *ortalama durum* 2

AVL tree → *AVL ağacı* 10

B

B+ tree → *B+ ağacı* 10

B-tree → *B-ağacı* 10

back edge → *geri kenar* 14

backtracking → *geri izleme* 4

balance factor → *denge faktörü* 10

balanced parentheses → *parantez kontrolü* 7

base case → *taban durum* 4

best case → *en iyi durum* 2

Big O → *büyük O* 2

Big Omega → *büyük Omega* 2

Big Theta → *büyük Teta* 2

big-O notation → *büyük O gösterimi* 3

binary heap → *ikili heap* 11

binary recursion → *ikili özyineleme* 4

binary search → *ikili arama* 2

binary search tree (BST) → *ikili arama ağacı* 10

binary tree → *ikili ağaç* 9

binomial heap → *binom heap* 11

bipartite graph → *iki parçalı çizge* 14

bit (binary digit) → *bit* 1

bit mask → *maske* 13

bit vector → *bit vektörü* 13

black height → *siyah yüksekliği* 10

bottom-up heap construction → *aşağıdan yukarıya heap oluşturma* 11

bounded stack → *sınırlı yığın* 7

breadth-first search (BFS) → *genişlik öncelikli arama* 14

brute force → *kaba kuvvet* 7

BST property → *BST özelliği* 10

bubble sort → *kabarcık sıralaması* 3

bucket → *kova* 12

buffer → *tampon* 8

byte → *bayt* 1

C

cache → *önbellek* 3

cache miss → *önbellek ıskası* 6

call stack → *çağrı yığını* 7

call tree → *çağrı ağacı* 4

capacity → *kapasite* 3

chain → *zincir* 12

character encoding → *karakter kodlaması* 1

child → *çocuk* 9

circular queue → *dairesel kuyruk* 8

circularly linked list → *dairesel bağlı liste* 5

client code → *istemci kodu* 1

code point → *kod noktası* 1

collection → *koleksiyon* 6

collision → *çakışma* 12

comparator → *karşılaştırıcı* 6

compile time → *derleme zamanı* 1

complete binary tree → *tam ikili ağaç* 9
complete graph → *tam çizge* 14
complexity → *karmaşıklık* 2
compression function → *sıkıştırma fonksiyonu* 12
connected → *bağlantılı* 14
connected component → *bağlantılı bileşen* 14
contiguous → *ardışık* 3
contract (equals/hashCode) → *sözleşme* 12
correctness → *doğruluk* 2
covariance → *kovaryans* 3
covariant → *kovaryant* 3
cursor → *imleç* 6
cycle → *döngü* *(çizgede: aynı düğümde başlayıp biten yol (bkz. loop))* 14
cycle detection → *döngü tespiti* 5

D

d-ary heap → *d'li heap* 11
data structure → *veri yapısı* 1
data type → *veri tipi* 1
decision tree → *karar ağacı* 4
deep copy → *derin kopya* 3
degenerate tree → *devrik ağaç* 10
degree → *derece* 9
dense graph → *yoğun çizge* 14
depth → *derinlik* 9
depth-first search (DFS) → *derinlik öncelikli arama* 14
deque (double-ended queue) → *çift uçlu kuyruk* 8
dequeue → *dequeue (önden çıkarma)* 8
descendant → *soy* 9
deterministic → *belirleyici* 12
difference → *fark* 13
digital data → *dijital veri* 1
Dijkstra's algorithm → *Dijkstra algoritması* 14
direct addressing → *doğrudan adresleme* 12
directed acyclic graph (DAG) → *yönlü döngüsüz çizge* 14
directed cycle → *yönlü döngü* 14
directed graph (digraph) → *yönlü çizge* 14
disjoint sets (union-find) → *ayrık kümeler* 13
double hashing → *çift hash* 12
doubling experiment → *katlama deneyi* 2
doubly linked list → *çift yönlü bağlı liste* 5
down-heap → *down-heap (aşağı süzme)* 11
dynamic array → *dinamik dizi* 3
dynamic size stack → *dinamik boyutlu yığın* 7

E

edge → *kenar* 9
edge list → *kenar listesi* 14
element → *öğe* 3
empirical analysis → *deneysel analiz* 2
enqueue → *enqueue (arkaya ekleme)* 8
Euler tour → *Euler turu* 9
exception → *istisna* 7
expression tree → *ifade ağacı* 9
external node → *dış düğüm* 9

F

fail-fast → *hızlı başarısız* 6
Fibonacci heap → *Fibonacci heap* 11
FIFO → *ilk giren ilk çıkar* 8
first-in, last-out (FILO) → *ilk giren son çıkar* 7
fixed size stack → *sabit boyutlu yığın* 7
floating point → *kayan nokta* 1
Floyd's cycle detection algorithm → *Floyd'un döngü tespit algoritması* 5
forest → *orman* 9
front → *ön* 8
full binary tree → *dolu ikili ağaç* 9

G

garbage collector → *çöp toplayıcı* 1
generic → *jenerik* 1
generics → *jenerikler* 1
graph → *çizge* 14

H

half-open interval → *yarı açık aralık* 6
hash code → *hash kodu* 12
hash function → *hash fonksiyonu* 12
hash set → *hash kümesi* 13
hash table → *hash tablosu* 12
head → *baş* 5
heap → *heap* 11
heap memory → *heap bellek* 3
heap sort → *heap sort* 11
heap-order property → *heap-sıra özelliği* 11
height → *yükseklik* 9
home slot → *ev yuvası* 12
Horner's method → *Horner yöntemi* 12

I

immutability → *değiştirilemezlik* 1
immutable → *değiştirilemez* 6
implementation → *gerçekleme* 1
in-degree → *giriş derecesi* 14
in-place → *yerinde* 3
incident → *bağlı (kenar ile düğüm)* 14
index → *indis* 3
infix → *ara ek* 7
initializer → *ilklendirici* 3
inorder → *kök ortada* 9
insertion sort → *eklemeli sıralama* 3

interface → *arayüz* 1
internal node → *iç düğüm* 9
intersection → *kesişim* 13
invariant → *değişmez* 2
inverse Ackermann function → *ters Ackermann fonksiyonu* 13
inversion → *ters sıralı çift* 3
isolated vertex → *yalnız düğüm* 14
iterator → *yineleyici* 6

J

jagged array → *düzensiz dizi* 3
Java Collections Framework → *Java Koleksiyon Çerçevesi* 6

K

Kahn's algorithm → *Kahn algoritması* 14
key → *anahtar* 11
key-value pair → *anahtar-değer çifti* 12
Kruskal's algorithm → *Kruskal algoritması* 14

L

lambda expression → *lambda ifadesi* 6
leaf → *yaprak* 9
level → *düzey* 9
level-order → *düzey sırası* 9
LIFO → *son giren ilk çıkar* 7
linear probing → *doğrusal yoklama* 12
linear recursion → *doğrusal özyineleme* 4
linear search → *doğrusal arama* 3
linked list → *bağlı liste* 5
list → *liste* 6
listener → *olay dinleyicisi* 6
load factor → *yük faktörü* 12
location-aware entry → *konum farkında öğe* 11
loitering → *oyalanma* 7
loop → *döngü* *(programda: for/while tekrar yapısı (bkz. cycle))* 2
loop invariant → *döngü değişmezi* 2
lower bound → *alt sınır* 2

M

MAD method (Multiply-Add-and-Divide) → *MAD yöntemi (çarp-topla-böl)* 12
map → *eşleme* 12
mathematical induction → *tümevarım* 4
matrix → *matris* 3
max-heap → *max-heap* 11
median → *ortanca* 3
membership test → *üyelik testi* 13
memoization → *bellekleme* 4
memory leak → *bellek sızıntısı* 3
merge sort → *birleştirme sıralaması* 3
min-heap → *min-heap* 11
minimum spanning tree (MST) → *minimum kapsayan ağaç* 14
mode → *mod* 3
monotonic stack → *monoton yığın* 7
multidimensional array → *çok boyutlu dizi* 3
multimap → *çoklu eşleme* 13
multiple recursion → *çoklu özyineleme* 4
multiplicity → *çokluk* 13
multiset → *çoklu küme* 13
mutable → *değiştirilebilir* 6

N

next greater element → *sonraki büyük öğe* 7
node → *düğüm* *(bağlı liste, ağaç ve çizgede; çizgede vertex ile aynı kavram)* 5
non-primitive → *ilkel olmayan* 1
null → *null* 5

O

off-by-one error → *sınır hatası* 3
open addressing → *açık adresleme* 12
operand → *işlenen* 7
operator → *işleç* 7
out-degree → *çıkış derecesi* 14
overflow → *taşma* 7
overload → *aşırı yükleme* 6

P

parent → *ebeveyn* 9
pass by value → *değerle geçirme* 1
path → *yol* 9
path compression → *yol kısaltma* 13
perfect binary tree → *mükemmel ikili ağaç* 9
perfect hash → *mükemmel hash* 12
persistent data structure → *kalıcı veri yapısı* 6
pointer → *işaretçi* 5
pop → *pop (üstten çıkarma)* 7
position → *konum* 6
positional list → *konumsal liste* 6
postfix → *sonek* 7
postorder → *kök sonda* 9
predecessor → *öncül* 10
prefix → *önek* 7
prefix averages → *önek ortalamaları* 2
preorder → *kök önde* 9
Prim's algorithm → *Prim algoritması* 14
primary clustering → *birincil kümelenme* 12
primitive operation → *ilkel işlem* 2
primitive type → *ilkel tip* 1
priority queue → *öncelikli kuyruk* 11

probe sequence → *yoklama dizisi* 12
probing → *yoklama* 12
pruning → *budama* 10
push → *push (üste ekleme)* 7

Q

quadratic probing → *karesel yoklama* 12
queue → *kuyruk* 8
quick sort → *hızlı sıralama* 3
Quick-Find → *Quick-Find* 13

R

race condition → *yarış durumu* 6
random access → *rastgele erişim* 3
range query → *aralık sorgusu* 10
rank → *rütbe* 13
rear → *arka* 8
recurrence relation → *yineleme bağıntısı* 4
recursion → *özyineleme* 4
recursive step → *özyinelemeli adım* 4
red-black tree → *kırmızı-siyah ağaç* 10
reference → *referans* 1
relaxation → *gevşetme* 14
resizing (rehashing) → *yeniden boyutlandırma* 12
return address → *dönüş adresi* 4
reverse Polish notation (RPN) → *ters Polonya gösterimi* 7
ring buffer → *halka tampon* 8
robustness → *sağlamlık* 2
root → *kök* 9
rotation → *döndürme* 10
round robin → *round robin (sırayla dönen çizelgeleme)* 8
run time → *çalışma zamanı* 1

S

scheduler → *çizelgeleyici* 8
search tree → *arama ağacı* 10
secondary clustering → *ikincil kümelenme* 12
selection sort → *seçimli sıralama* 3
self-balancing search tree → *kendini dengeleyen arama ağacı* 10
self-loop → *öz döngü* 14
self-referential structure → *öz-referanslı yapı* 5
sentinel → *bekçi* 5
separate chaining → *ayrı zincirleme* 12
set → *küme* 13
shallow copy → *sığ kopya* 3
short-circuit evaluation → *kısa devre* 3
shortest path → *en kısa yol* 14
sibling → *kardeş* 9
simple path → *basit yol* 9
simple uniform hashing → *basit düzgün hash* 12
single-source shortest path → *tek kaynaklı en kısa yol* 14
singly linked list → *tek yönlü bağlı liste* 5
skewed tree → *çarpık ağaç* 9
sliding window → *kayan pencere* 8
slot → *yuva* 12
slow and fast pointers → *yavaş ve hızlı işaretçi* 5
sorted list → *sıralı liste* 11
space complexity → *alan karmaşıklığı* 2
spanning tree → *kapsayan ağaç* 14
sparse graph → *seyrek çizge* 14
splay tree → *splay ağacı* 10
stable (sort) → *kararlı* 3
stack → *yığın* 7
stack frame → *yığın çerçevesi (çağrı yığınında bir metot çağrısının kaydı)* 4
stack trace → *yığın izi* 4
stale entry → *bayat kayıt* 14
starvation → *açlık* 11
static array → *statik dizi* 3
stock span → *stock span* 7
stream → *akış* 6
structural modification → *yapısal değişiklik* 6
subset → *alt küme* 13
subtree → *alt ağaç* 9
successor → *ardıl* 10
swap → *takas* 3

T

tail → *son* 5
tail call → *kuyruk çağrısı* 4
tail call optimization → *kuyruk çağrısı eniyilemesi* 4
tail pointer → *son işaretçisi* 5
tail recursion → *kuyruk özyinelemesi* 4
tentative distance → *geçici mesafe* 14
thread → *iş parçacığı* 6
tight bound → *sıkı sınır* 2
time complexity → *zaman karmaşıklığı* 2
time slice (quantum) → *zaman dilimi* 8
TimSort → *TimSort* 6
tombstone (DEFUNCT) → *silindi işareti* 12
top → *üst* 7
topological sort → *topolojik sıralama* 14
Towers of Hanoi → *Hanoi kuleleri* 4
trade-off → *ödünleşim* 1
transpose → *transpoz* 3
traversal → *gezinme* 5
tree → *ağaç* 9
two-dimensional array → *iki boyutlu dizi* 3
type parameter → *tür parametresi* 1
type safety → *tip güvenliği* 1

U

unboxing → *kutudan çıkarma* 6
underflow → *alt taşma* 7
undirected graph → *yönsüz çizge* 14
undo → *geri al* 7
Unicode → *Unicode* 1
union → *birleşim* 13
union by rank → *rütbeye göre birleştirme* 13
union by size → *boyuta göre birleştirme* 13
universe → *evren* 13
unsorted list → *sırasız liste* 11
unwinding → *çözülme* 4
up-heap → *up-heap (yukarı süzme)* 11
upper bound → *üst sınır* 2

V

value → *değer* 12
varargs → *değişken sayıda argüman* 3
vertex → *düğüm (çizgede; node ile aynı karşılık (bkz. node))* 14
view → *görünüm* 6

W

weight → *ağırlık* 14
weighted graph → *ağırlıklı çizge* 14
worst case → *en kötü durum* 2
wrapper class → *sarmalayıcı sınıf* 6

Z

zig, zig-zig, zig-zag → *zig, zig-zig, zig-zag* 10

APPENDIX B

Prerequisites for Data Structures

Data structures draw on programming, abstraction, algorithms, and mathematical reasoning all at once. A working knowledge of Java, references, arrays, recursion, complexity, and basic discrete mathematics gives you solid ground to build on.

The essentials

The best preparation is hands-on programming experience. Beyond that, you should be able to trace code step by step, understand how references behave, and compare what different operations cost.

Suggested order of preparation

Programming **Java objects** **References** **Arrays** **Recursion** **Discrete math** **Big O** **Tracing and debugging**

Core prerequisites

Programming Fundamentals

Variables · control flow · methods

You need to be comfortable with basic programming before implementing a real data structure.

WHY IT MATTERS

A data structure is defined not only by the values it stores but by the operations it offers and the invariants its code maintains.

- Trace assignments, conditionals, loops, and method calls step by step.
- Write methods that take parameters and return values.
- Understand local variables and scope.
- Read nested loops and estimate how many times they run.

Where it shows up: Traversal, insertion, deletion, search, and update operations.

Variables **Loops** **Methods**

Java Classes and Objects

Objects · classes · constructors

The course implements everything in Java, so classes, objects, fields, constructors, and methods should already be familiar.

WHY IT MATTERS

Most data structures are implemented as classes whose internal representation stays behind their operations.

- Tell a class from an object.
- Create objects with new and use constructors.
- Understand instance fields, local variables, and the this keyword.
- Know encapsulation, access modifiers, and the difference between static and instance members.
- Throw and catch exceptions, for example when someone tries to remove an element from an empty structure.
- Read interfaces and generic types such as List<Integer>; Chapter 1 revisits both.

Where it shows up: Node classes, container classes, ADTs, and encapsulated invariants.

Java **Classes** **Objects**

References and Aliasing

References · null · aliasing

Linked structures rely on references that connect objects at run time.

WHY IT MATTERS

Understanding references is essential for linked lists, trees, graphs, and object-based structures.

- Know that an object variable holds a reference to the object, not a copy of it.
- Recognize that two variables can refer to the same object (aliasing).
- Use null safely and know what it represents.
- See the difference between modifying an object and assigning a new value to a reference.

Where it shows up: Next-node links, child references, graph vertices, and structural updates.

References **null** **Aliasing**

Arrays

Indexed storage · contiguous positions

Arrays underlie dynamic arrays, heaps, hash tables, adjacency matrices, and many internal representations. Basic array use is assumed. Chapter 3 covers arrays and array algorithms in detail; how ArrayList grows as a dynamic array is in Chapter 6.

WHY IT MATTERS

Constant-time access by index makes the array one of the most important building blocks of data structure design.

- Use zero-based indexing correctly.
- Traverse an array and tell an index from the value stored there.
- Declare, fill, and print one- and two-dimensional arrays.
- Avoid out-of-bounds indices and know when ArrayIndexOutOfBoundsException is thrown.

Where it shows up: ArrayList, binary heaps, hash table buckets, and adjacency matrices.

Arrays **Indexing** **Memory**

Recursion

Base case · recursive case · call stack

Tree and graph algorithms are usually written recursively. Chapter 4 revisits recursion before trees and graphs put it to use.

WHY IT MATTERS

Recursion is the natural way to process hierarchical structures: solve the same problem on each substructure.

- Identify the base case that ends the recursion.
- Trace recursive calls and returns.
- Understand stack depth and infinite recursion.
- Recognize recursive patterns in tree traversal and depth-first search.

Where it shows up: Tree traversals, DFS, recursive insertion and deletion, and divide and conquer.

Recursion **Trees** **Call stack**

Big-O Notation

Growth rates · upper bounds · scalability

Data structures are compared by how the cost of their operations grows with the input. Chapter 2 introduces the notation formally; knowing the common growth rates beforehand makes that chapter easier.

WHY IT MATTERS

Big O is the language for explaining why one representation scales better than another.

- Recognize O(1), O(log n), O(n), O(n log n), and O(n^2).
- Estimate the cost of nested loops.
- Understand why repeated halving gives logarithmic behavior.
- Weigh time against space instead of relying on a single measure.

Where it shows up: Cost tables for arrays, lists, trees, heaps, hash tables, and graphs.

Big O **Complexity** **Scalability**

Tracing Algorithms

State changes · invariants · dry runs

Tracing an algorithm by hand is one of the most useful skills for understanding a data structure.

WHY IT MATTERS

A trace shows how pointers, indices, stacks, queues, and tree links change from one step to the next.

- Follow the state of the variables and the structure after every operation.
- Trace loops and recursive calls in execution order.
- Check whether structural invariants still hold after an update.
- Think through edge cases on small examples.

Where it shows up: Using the simulators, debugging, checking correctness, and reasoning through exam questions.

Tracing **Invariants** **Dry runs**

Basic Discrete Mathematics

Sets · relations · logarithms

A little discrete mathematics makes it easier to reason about sets, trees, graphs, and complexity.

WHY IT MATTERS

Mathematical notation lets you state relationships, bounds, connectivity, and structural properties precisely.

- Be comfortable with sets, membership, union, intersection, and difference.
- Understand simple relations and connections such as those in a graph.
- Use exponents and logarithms when thinking about balanced trees and repeated halving.
- Read simple sums and inequalities when comparing algorithm costs.

Where it shows up: Sets, graph connectivity, tree height, and asymptotic analysis.

Sets **Relations** **Logarithms**

Induction and Proof Techniques

Induction · contradiction · divisibility

Many results in the course, such as bounds on tree height, the correctness of recursive algorithms, and probe sequences in hash tables, are established with short proofs, mostly by induction or contradiction.

WHY IT MATTERS

A proof shows that a structure keeps its promised bound on every input; trying a few examples cannot give that guarantee.

- Set up the base case and the inductive step, and see where the inductive hypothesis is used.
- Follow an induction on a quantity such as the number of nodes, the height, or the rank.
- Recognize proof by contradiction: assume the opposite and derive a contradiction.
- Know divisibility, prime numbers, and common divisors.

Where it shows up: Correctness of recursion, relations between node count and height in binary trees, height bounds for red-black trees and disjoint-set trees, and why quadratic probing and double hashing try distinct slots.

Induction **Contradiction** **Divisibility**

Debugging and Testing

Assertions · edge cases · inspecting intermediate states

Data structure bugs usually come from wrong links, wrong indices, boundary conditions, or broken invariants.

WHY IT MATTERS

Systematic debugging is more reliable than changing code by trial and error until the output looks right.

- Test empty, single-element, and boundary cases.
- Look at intermediate states, not just the final output.

- Use assert statements or explicit checks for invariants.
- Separate bad input from logic errors in the structure.

Where it shows up: Linked list updates, rotations, heap order, hash table resizing, and graph traversal.

Debugging **Testing** **Edge cases**

APPENDIX C

Important People in Data Structures

Many of the structures and algorithms in this course still carry the names of the people who discovered them: Dijkstra's algorithm, the AVL tree, Kruskal's algorithm. The profiles below follow the order of the course and tie each contribution to the chapter where it is covered.

From arrays and linked lists to sets and graphs

These are not biographies. The aim is to connect each name to a concrete structure, algorithm, invariant, or idea about performance.

Key contributions

Niklaus Wirth

Programs as algorithms on data

Designed the Pascal language and wrote Algorithms + Data Structures = Programs (1976).

WHY IT MATTERS

The title of the book sums up the central idea of this course: a program is an algorithm working on a well-chosen data structure.

In the course: Chapter 1 · Introduction

Pascal **Algorithms** **Data structures**

Barbara Liskov

Abstract data types

Introduced the abstract data type (ADT) with Stephen Zilles in 1974 and made it the foundation of the CLU language. She received the 2008 Turing Award.

WHY IT MATTERS

An ADT separates what the operations do from how they are implemented, which is exactly how the course defines stacks, queues, lists, maps, and sets.

In the course: Chapter 1 · Introduction

ADT **Interfaces** **Encapsulation**

Donald E. Knuth

Analysis of algorithms

Wrote The Art of Computer Programming and made the mathematical analysis of algorithms standard practice, together with O, Ω, and Θ notation.

WHY IT MATTERS

His systematic treatment of searching, sorting, trees, and hashing turned implementation techniques into an engineering discipline. The expected probe counts for linear probing are also his results.

In the course: Chapter 2 · Algorithm Analysis, Chapter 12 · Maps and Hash Tables

Big O **Analysis** **Hashing**

John von Neumann

Sorting · Merge sort

Described merge sort in 1945, making it one of the first algorithms written for a stored-program computer.

WHY IT MATTERS

Merge sort is the standard example of divide and conquer and of a stable O(n log n) sort.

In the course: Chapter 3 · Sorting Algorithms

Merge sort **Divide and conquer** **Stable sort**

C. A. R. Hoare

Sorting · Quicksort

Developed quicksort in 1959–1961 and later created Hoare logic for proving programs correct.

WHY IT MATTERS

Quicksort brings together partitioning, recursion, in-place rearrangement, and the effect of the pivot choice on average and worst-case cost.

In the course: Chapter 3 · Sorting Algorithms

Quicksort **Partitioning** **Recursion**

Allen Newell, Cliff Shaw, and Herbert A. Simon

Linked lists

First used linked lists in the Information Processing Language (IPL), which they developed in 1955–1956.

WHY IT MATTERS

Nodes connected by references made structures that grow and shrink at run time possible. Linked lists, linked stacks, queues, and trees all build on this idea.

In the course: Chapter 5 · Linked Lists

Linked list **Nodes** **References**

Robert W. Floyd

Cycle detection · Heap construction

The tortoise-and-hare cycle detection algorithm is attributed to him. His Treesort 3 algorithm (1964) builds a heap bottom-up.

WHY IT MATTERS

Two pointers moving at different speeds find a cycle in O(n) time with O(1) extra space. Bottom-up heap construction takes O(n) instead of O(n log n).

In the course: Chapter 5 · Linked Lists, Chapter 11 · Priority Queues and Heaps

Cycle detection **Two pointers** **Heap construction**

Joshua Bloch

Java Collections Framework

Designed the Java Collections Framework introduced in Java 1.2 (1998).

WHY IT MATTERS

The List, Set, Map, and Iterator interfaces, with classes such as ArrayList, LinkedList, and HashMap, separate the ADT from the data structure that implements it, just as the course does.

In the course: Chapter 6 · The List ADT and Java Collections

List **Iterator** **Java collections**

Friedrich L. Bauer and Klaus Samelson

Stacks

Formulated the stack principle (Kellerprinzip, 1955–1957) for translating and evaluating arithmetic formulas.

WHY IT MATTERS

The last-in, first-out (LIFO) stack became the standard tool for expression evaluation, compilers, and managing method calls.

In the course: Chapter 7 · Stacks

Stack **LIFO** **Expressions**

Jan Łukasiewicz

Prefix notation

Introduced Polish notation, that is, prefix notation, in the 1920s. Reverse Polish notation, or postfix notation, is derived from it.

WHY IT MATTERS

Prefix and postfix expressions need no parentheses and can be evaluated with a single stack in one pass.

In the course: Chapter 7 · Stacks

Prefix **Postfix** **Notation**

Edsger W. Dijkstra

Shunting-yard · Shortest paths

Developed the shunting-yard algorithm (1961), which converts infix expressions to postfix, and the shortest-path algorithm for weighted graphs (1959).

WHY IT MATTERS

Shunting-yard works with an operator stack and precedence rules. The shortest-path algorithm combines relaxation with a priority queue that returns the element with the smallest key.

In the course: Chapter 7 · Stacks, Chapter 14 · Graphs

Infix → postfix **Shortest path** **Priority queue**

Georgy Adelson-Velsky and Evgenii Landis

AVL trees

Introduced the AVL tree, the first self-balancing binary search tree, in 1962.

WHY IT MATTERS

A balance factor and single or double rotations keep the height at O(log n), so search, insertion, and deletion stay logarithmic.

In the course: Chapter 10 · Search Trees

AVL **Balance factor** **Rotations**

Leonidas J. Guibas and Robert Sedgewick

Red-black trees

Introduced the red-black tree in 1978, building on Rudolf Bayer's symmetric binary B-trees (1972).

WHY IT MATTERS

Node colors and a few rotations keep the tree balanced. Java's TreeMap and TreeSet are implemented as red-black trees.

In the course: Chapter 10 · Search Trees

Red-black tree **Balancing** **TreeMap**

Daniel D. Sleator and Robert E. Tarjan

Splay trees

Introduced the splay tree in 1985.

WHY IT MATTERS

A splay tree keeps no balance information. It moves every accessed node to the root, which makes operations run in amortized O(log n) time and speeds up access to recently used keys.

In the course: Chapter 10 · Search Trees

Splay tree **Amortized analysis** **Locality**

Rudolf Bayer and Edward M. McCreight

B-trees

Developed the B-tree at Boeing Scientific Research Labs (1970; published 1972).

WHY IT MATTERS

A high fan-out, with many children per node, keeps the tree shallow, so each search reads only a few disk blocks. B-trees and B+ trees are the basis of database and file-system indexes.

In the course: Chapter 10 · Search Trees

B-tree **B+ tree** **Indexing**

J. W. J. Williams

Binary heap · Heapsort

Introduced the binary heap and the heapsort algorithm in 1964.

WHY IT MATTERS

A complete binary tree stored in an array supports insertion and removal of the minimum (or maximum) in O(log n) time. The same structure also gives an in-place O(n log n) sort.

In the course: Chapter 11 · Priority Queues and Heaps

Binary heap **Heapsort** **Priority queue**

Michael L. Fredman and Robert E. Tarjan

Fibonacci heaps

Introduced the Fibonacci heap in 1984.

WHY IT MATTERS

Its amortized O(1) decrease-key operation shows how a specialized heap can speed up graph algorithms such as shortest paths.

In the course: Chapter 11 · Priority Queues and Heaps

Fibonacci heap **Amortized** **Priority queue**

Hans Peter Luhn

Hashing

Proposed hashing with chaining in an internal IBM memorandum in 1953.

WHY IT MATTERS

Mapping a key directly to a bucket brings lookup down to expected O(1) time. Separate chaining is still one of the standard ways to resolve collisions.

In the course: Chapter 12 · Maps and Hash Tables

Hashing **Chaining** **Buckets**

Bernard A. Galler and Michael J. Fischer

Disjoint-set forests

Introduced the disjoint-set forest representation in 1964.

WHY IT MATTERS

Each set is a tree whose root is the set's representative, so union and find need nothing but parent links.

In the course: Chapter 13 · Sets and Disjoint Sets

Disjoint sets **Union-find** **Forests**

Robert E. Tarjan

Union-find analysis

Proved in 1975 that with union by rank and path compression together, m operations take $O(m \cdot \alpha(n))$ time.

WHY IT MATTERS

The inverse Ackermann function $\alpha(n)$ grows so slowly that each operation takes constant time for all practical purposes.

In the course: Chapter 13 · Sets and Disjoint Sets

Union by rank **Path compression** **Amortized**

Edward F. Moore

Breadth-first search

Published breadth-first search (BFS) in 1959 as a way to find the shortest path through a maze.

WHY IT MATTERS

BFS uses a queue to visit vertices level by level and finds shortest paths in unweighted graphs in $O(n + m)$ time.

In the course: Chapter 14 · Graphs

BFS **Queue** **Shortest path**

Arthur B. Kahn

Topological sorting

Published the topological sorting algorithm based on in-degrees in 1962.

WHY IT MATTERS

Repeatedly removing vertices of in-degree zero puts the vertices of a directed graph in topological order; if the graph is not a directed acyclic graph (DAG), the procedure exposes the cycle.

In the course: Chapter 14 · Graphs

Topological sort **DAG** **In-degree**

Joseph B. Kruskal

Minimum spanning trees

Published his minimum spanning tree (MST) algorithm in 1956.

WHY IT MATTERS

Kruskal's algorithm sorts the edges by weight and uses a disjoint-set (union-find) structure to reject edges that would form a cycle.

In the course: Chapter 14 · Graphs

MST **Greedy** **Union-find**

Robert C. Prim

Prim's algorithm

While at Bell Labs, published the minimum spanning tree algorithm that grows the tree from a single vertex, in “Shortest connection networks and some generalizations” (1957). Vojtěch Jarník (1930) and Edsger W. Dijkstra (1959) gave the same method independently.

WHY IT MATTERS

At each step Prim's algorithm adds the lightest edge between the tree and the vertices outside it; the cut property guarantees that this edge belongs to a minimum spanning tree. With the candidate edges kept in a min-priority queue, the algorithm runs in O(m log n) time.

In the course: Chapter 14 · Prim's Algorithm, Chapter 14 · Kruskal vs. Prim

MST **Cut property** **Priority queue**