Chapter 4: Linked Lists

Data Structures and Algorithms with C

Nodes, malloc, free | Insert at front · back · middle | Deletion and search

Reversing · merging | Floyd's cycle detection | Circular and doubly linked

Header nodes | The available space list

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/dsa_c/slides/Chapter_04_Linked_Lists.html

INTRODUCTION

Learning Outcomes

Building nodes

Define a self-referential `struct`, check the result of `malloc`, release with `free`.

Link discipline

Insert at the front, back, middle and in order, in the right sequence, without losing a link.

Safe deletion

Delete nodes without memory leaks or dangling pointers.

Classic routines

Write reversal, merging, finding the middle and Floyd's cycle detection with O(1) extra space.

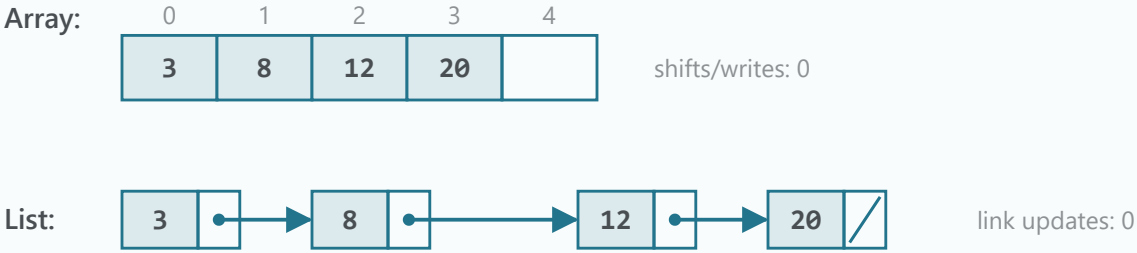
Kinds of lists

Choose circular, doubly linked and header-node lists to fit the task.

Memory management

Reuse nodes with an available space list.

When Arrays Fall Short: Links Instead of Shifting

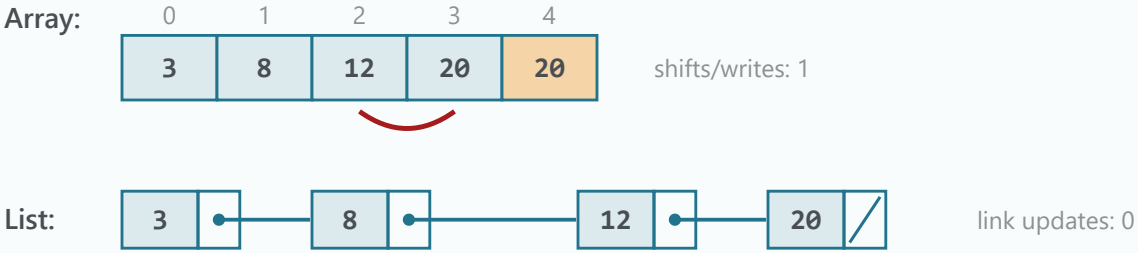


We insert **10 after 8**. The array must shift later elements to make room; the list only changes links.

Step 1 / 6

Instructor “If you don’t learn pointers now, you still won’t know them in your senior year.”

When Arrays Fall Short: Links Instead of Shifting

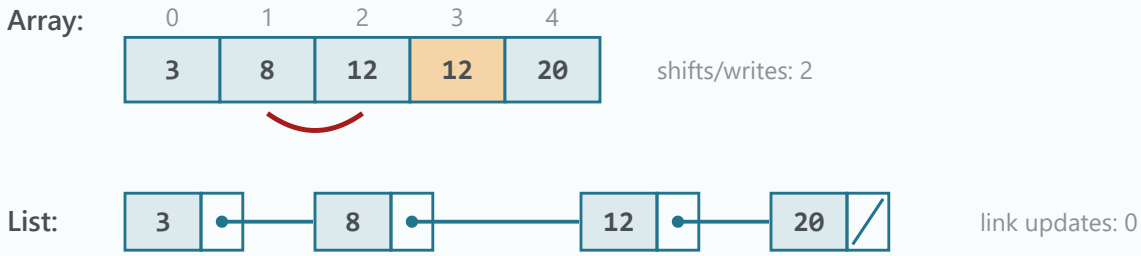


Array: 20 moved one place right (move 1).

Step 2 / 6

Instructor “If you don’t learn pointers now, you still won’t know them in your senior year.”

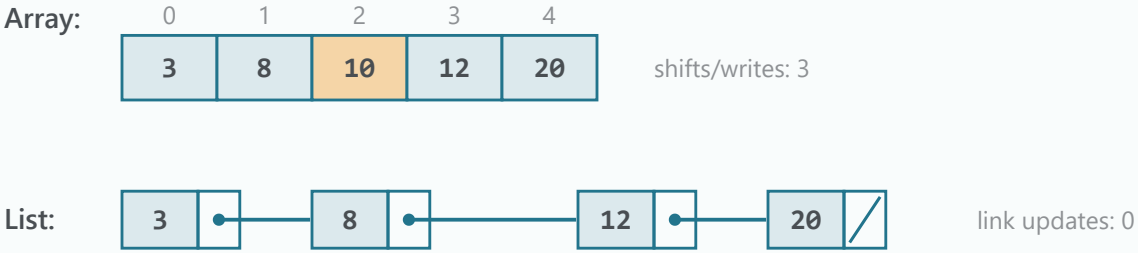
When Arrays Fall Short: Links Instead of Shifting



Array: 12 moved one place right (move 2). An n-element array needs n moves in the worst case: **O(n)**.

Instructor “If you don’t learn pointers now, you still won’t know them in your senior year.”

When Arrays Fall Short: Links Instead of Shifting

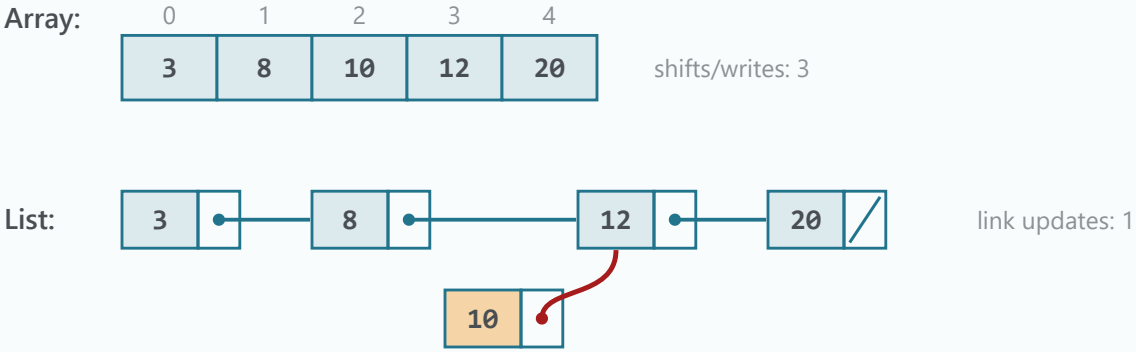


Array: 10 written into the freed cell.

Step 4 / 6

Instructor “If you don’t learn pointers now, you still won’t know them in your senior year.”

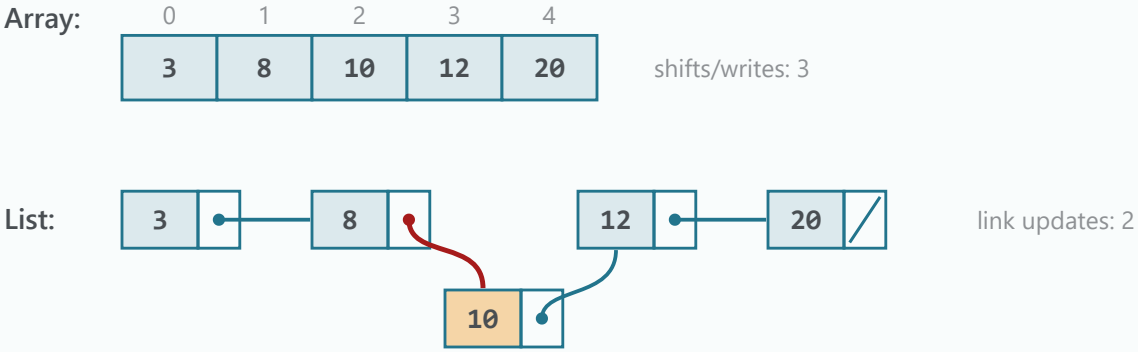
When Arrays Fall Short: Links Instead of Shifting



List: a new node is allocated and `temp→link` points to 12. No element moved.

Instructor “If you don’t learn pointers now, you still won’t know them in your senior year.”

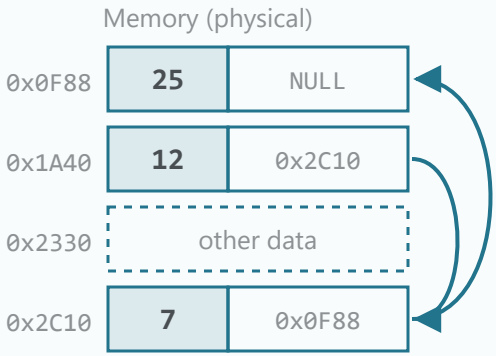
When Arrays Fall Short: Links Instead of Shifting



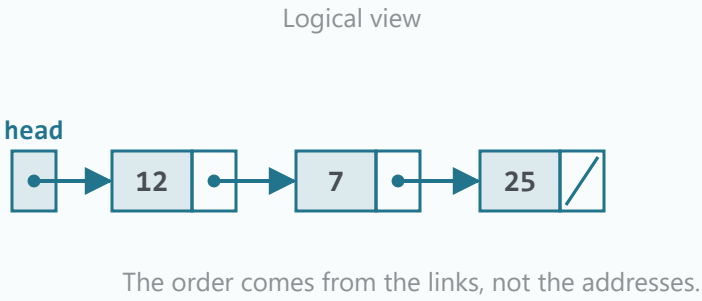
List: `8→link = temp`. If the position is known, just two links: **O(1)**.

Instructor “If you don’t learn pointers now, you still won’t know them in your senior year.”

Nodes Need Not Sit Next to Each Other in Memory



`head = 0x1A40`



The order comes from the links, not the addresses.

Node = data + link

The `link` field holds the **address** of the next node; `NULL` in the last node.

No random access

The address of node i cannot be computed; follow the links from the head: O(i).

What is an address?

On old x86 an address had two parts (segment + offset): segmented memory. Today it is a flat address space.

A Pointer Refresher

```
int x = 5;
int *p = &x;        // p holds the address of x
*p = 7;             // write through p: x == 7

node_ptr q = NULL; // no node yet
q = malloc(sizeof *q);
if (q != NULL) {
    q->data = 3;    // (*q).data = 3
    q->link = NULL;
}
free(q);            // released; q now dangles
q = NULL;
```

`&` and `*`

`&x` : the address of x · `*p` : the object p points to.

→

`q→data` is the same as `(*q).data` ; the parentheses are needed for precedence.

NULL

A pointer that points nowhere; it represents the empty list.

Instructor “We set the topic aside for two lectures to cover C and pointers. You may think you know pointers very well and still not follow; that is perfectly normal.”

Allocating a Node: `malloc`, the Check, `free`

```
node_ptr new_node(int x) {
    node_ptr p = malloc(sizeof *p);
    if (p == NULL) {
        fprintf(stderr, "No memory, no memoryyy...\n");
        exit(1);
    }
    p->data = x;
    p->link = NULL;
    return p;
}
```

`sizeof *p` and NULL

Right size without repeating the type. Out of memory, `malloc` returns NULL; sample message: *No memory, no memoryyy...*

The free rule

One `free` per `malloc`; double free or use after free is undefined behavior.

Instructor "Not every pointer needs a malloc. No such rule!"

Traversal pointers (`p = p->link`) point to existing nodes; no malloc.

A Node: A Self-Referential struct

```c
typedef struct node *node_ptr;  // pointer type first

struct node {
    int data;         // data field
    node_ptr link;    // address of the next node
};

node_ptr list = NULL;  // empty list
```

Why the typedef first?

The structure contains a pointer to its own type; declaring a pointer to an incomplete type is allowed.

Naming

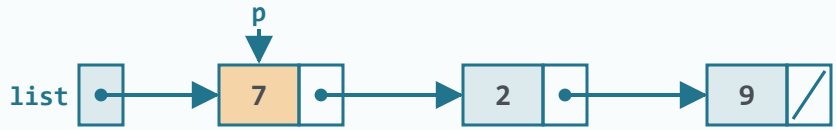
The pointer type can be called `list_pointer`, `node_ptr` or similar; here `node_ptr`.



struct node: data field + link field

Common mistake: writing `typedef struct list { … };` without a name for the typedef. Either drop the `typedef` or put a type name at the end.

Traversing a List: `p = p→link`



`print_list(list)` : the pointer is passed **by value**; advancing p does not change `list` in main.

Step 1 / 8

```
▶ void print_list(node_ptr p) {
2     for (; p != NULL; p = p->link)
3         printf("%d ", p->data);
4     printf("\n");
5 }
```

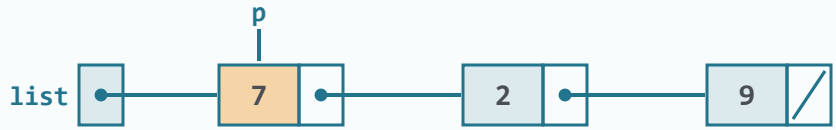
VARIABLES

`p = &node(7)`

OUTPUT

The length is counted the same way: `for (; p ≠ NULL; p = p→link) n++;`

Traversing a List: `p = p→link`



p is at the first node: print 7.

Step 2 / 8

```
1 void print_list(node_ptr p) {
2     for (; p != NULL; p = p->link)
          printf("%d ", p->data);
4     printf("\n");
5 }
```

VARIABLES

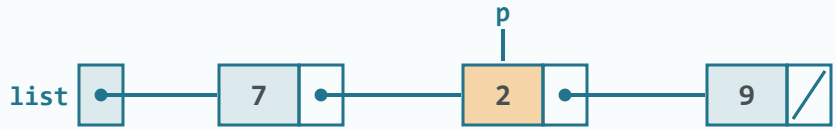
`p = &node(7)`

OUTPUT

```
7
```

The length is counted the same way: `for (; p ≠ NULL; p = p→link) n++;`

Traversing a List: `p = p→link`



`p = p→link` : the second node.

Step 3 / 8

```
1 void print_list(node_ptr p) {
▶     for (; p != NULL; p = p->link)
3         printf("%d ", p->data);
4     printf("\n");
5 }
```

VARIABLES

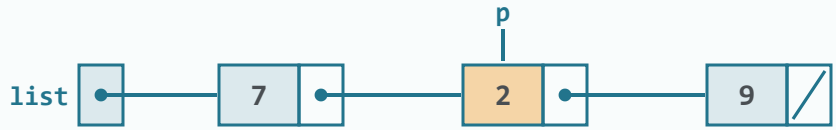
`p = &node(2)`

OUTPUT

```
7
```

The length is counted the same way: `for (; p ≠ NULL; p = p→link) n++;`

Traversing a List: `p = p→link`



Print 2.

Step 4 / 8

```
1 void print_list(node_ptr p) {
2     for (; p != NULL; p = p->link)
          printf("%d ", p->data);
4     printf("\n");
5 }
```

VARIABLES

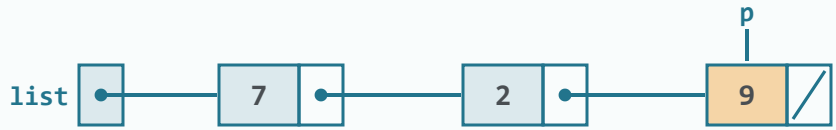
`p = &node(2)`

OUTPUT

```
7 2
```

The length is counted the same way: `for (; p ≠ NULL; p = p→link) n++;`

Traversing a List: `p = p→link`



The third node.

Step 5 / 8

```
1 void print_list(node_ptr p) {
▶     for (; p != NULL; p = p->link)
3         printf("%d ", p->data);
4     printf("\n");
5 }
```

VARIABLES

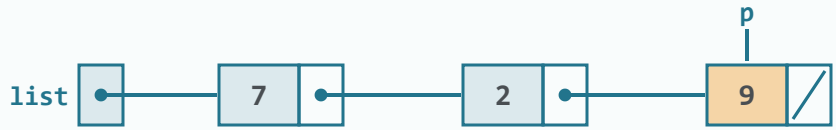
`p = &node(9)`

OUTPUT

```
7 2
```

The length is counted the same way: `for (; p ≠ NULL; p = p→link) n++;`

Traversing a List: `p = p→link`



Print 9.

Step 6 / 8

```
1 void print_list(node_ptr p) {
2     for (; p != NULL; p = p->link)
          printf("%d ", p->data);
4     printf("\n");
5 }
```

VARIABLES

`p = &node(9)`

OUTPUT

```
7 2 9
```

The length is counted the same way: `for (; p ≠ NULL; p = p→link) n++;`

Traversing a List: `p = p→link`

`p = NULL`



`p == NULL` : the loop ends.

Step 7 / 8

```
1 void print_list(node_ptr p) {
▶     for (; p != NULL; p = p->link)
3         printf("%d ", p->data);
4     printf("\n");
5 }
```

VARIABLES

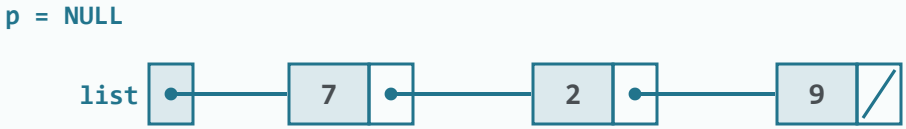
`p = NULL`

OUTPUT

```
7 2 9
```

The length is counted the same way: `for (; p ≠ NULL; p = p→link) n++;`

Traversing a List: `p = p→link`



End of line. n rounds for n nodes: **O(n)**.

Step 8 / 8

```
1 void print_list(node_ptr p) {
2     for (; p != NULL; p = p->link)
3         printf("%d ", p->data);
▶     printf("\n");
5 }
```

VARIABLES

`p = NULL`

OUTPUT

```
7 2 9
```

The length is counted the same way: `for (; p ≠ NULL; p = p→link) n++;`

Inserting at the Front, and Passing by Address

```
bool insert_front(node_ptr *head, int x) {
    node_ptr temp = malloc(sizeof *temp);
    if (temp == NULL) return false;
    temp->data = x;
    temp->link = *head;
    *head = temp;
    return true;
}

node_ptr list = NULL;
insert_front(&list, 2);   // 2
insert_front(&list, 7);   // 7 -> 2
```

Why `node_ptr *head`?

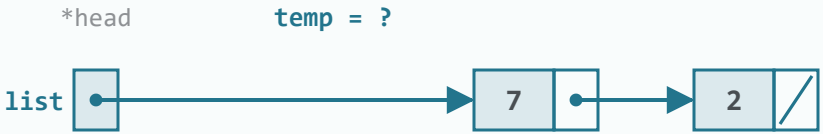
The head will change. Passed by value, only a local copy would change and `list` in main would stay stale.

Order matters

Writing `*head = temp` first would link the node to itself and lose the rest of the list. Cost **O(1)**.

Instructor "Normally, a variable that appears as '*param' in a function's formal parameter list should be used as '*param' inside the function too."

Code Trace: `insert_front(&list, 5)`



The call: `insert_front(&list, 5)`. `head` is the **address** of the variable `list` in main; the list is 7 → 2.

Step 1 / 7

```
▶ bool insert_front(node_ptr *head, int x) {
2     node_ptr temp = malloc(sizeof *temp);
3     if (temp == NULL) return false;  // no memory
4     temp->data = x;
5     temp->link = *head;    // first keep old head
6     *head = temp;          // then move the head
7     return true;
8 }
```

VARIABLES

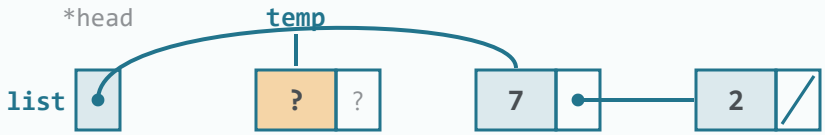
`head = &list` `x = 5` `temp = ?` `*head = &node(7)`

CALL STACK

`insert_front(&list, 5)`

`main()`

Code Trace: `insert_front(&list, 5)`



`malloc` allocated a node on the heap. Its fields still hold **garbage**.

Step 2 / 7

```
1 bool insert_front(node_ptr *head, int x) {
▶     node_ptr temp = malloc(sizeof *temp);
3     if (temp == NULL) return false;  // no memory
4     temp->data = x;
5     temp->link = *head;    // first keep old head
6     *head = temp;          // then move the head
7     return true;
8 }
```

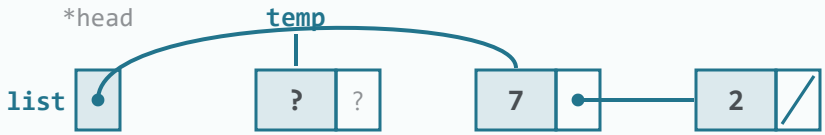
VARIABLES

`head = &list` `x = 5` `temp = &node(?)` `*head = &node(7)`

CALL STACK

`insert_front(&list, 5)`
`main()`

Code Trace: `insert_front(&list, 5)`



`temp ≠ NULL` : allocation succeeded, go on. (Otherwise it would return `false` with the list intact.)

Step 3 / 7

```
1 bool insert_front(node_ptr *head, int x) {
2     node_ptr temp = malloc(sizeof *temp);
▶     if (temp == NULL) return false;  // no memory
4     temp->data = x;
5     temp->link = *head;    // first keep old head
6     *head = temp;          // then move the head
7     return true;
8 }
```

VARIABLES

`head = &list` `x = 5`

`temp = &node(?)`

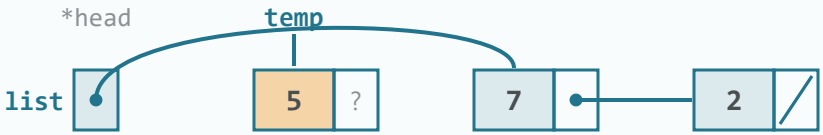
`*head = &node(7)`

CALL STACK

`insert_front(&list, 5)`

`main()`

Code Trace: `insert_front(&list, 5)`



Fill the data field: `temp→data = 5`.

Step 4 / 7

```
1 bool insert_front(node_ptr *head, int x) {
2     node_ptr temp = malloc(sizeof *temp);
3     if (temp == NULL) return false;  // no memory
▶     temp->data = x;
5     temp->link = *head;    // first keep old head
6     *head = temp;          // then move the head
7     return true;
8 }
```

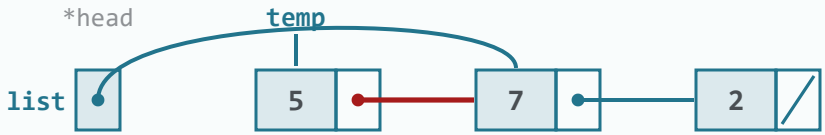
VARIABLES

`head = &list` `x = 5` `temp = &node(5)` `*head = &node(7)`

CALL STACK

`insert_front(&list, 5)`
`main()`

Code Trace: `insert_front(&list, 5)`



First keep the old head: `temp->link = *head` → the new node points to 7. `list` still points to 7.

Step 5 / 7

```
1 bool insert_front(node_ptr *head, int x) {
2     node_ptr temp = malloc(sizeof *temp);
3     if (temp == NULL) return false;  // no memory
4     temp->data = x;
▶     temp->link = *head;    // first keep old head
6     *head = temp;          // then move the head
7     return true;
8 }
```

VARIABLES

`head = &list` `x = 5`

`temp = &node(5)`

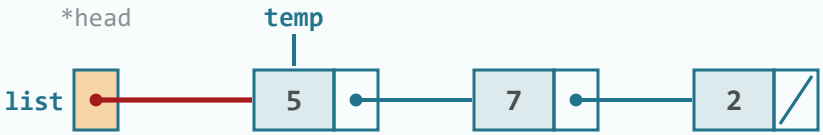
`*head = &node(7)`

CALL STACK

`insert_front(&list, 5)`

`main()`

Code Trace: `insert_front(&list, 5)`



Then change the head: `*head = temp`. Thanks to passing by address, `list` in main is updated.

Step 6 / 7

```
1 bool insert_front(node_ptr *head, int x) {
2     node_ptr temp = malloc(sizeof *temp);
3     if (temp == NULL) return false;  // no memory
4     temp->data = x;
5     temp->link = *head;    // first keep old head
▶     *head = temp;          // then move the head
7     return true;
8 }
```

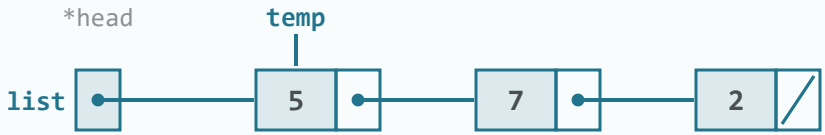
VARIABLES

`head = &list` `x = 5` `temp = &node(5)` `*head = &node(5)`

CALL STACK

`insert_front(&list, 5)`
`main()`

Code Trace: `insert_front(&list, 5)`



The list is now 5 → 7 → 2. A constant amount of work, independent of the length: **O(1)**.

Step 7 / 7

```
1 bool insert_front(node_ptr *head, int x) {
2     node_ptr temp = malloc(sizeof *temp);
3     if (temp == NULL) return false;  // no memory
4     temp->data = x;
5     temp->link = *head;    // first keep old head
6     *head = temp;          // then move the head
▶     return true;
8 }
```

VARIABLES

`head = &list` `x = 5` `temp = &node(5)` `*head = &node(5)`

CALL STACK

`insert_front(&list, 5)`

`main()`

Inserting at the Back: Finding the Last Node Is O(n)



The new node (5) is ready; `link = NULL` since it will be last.

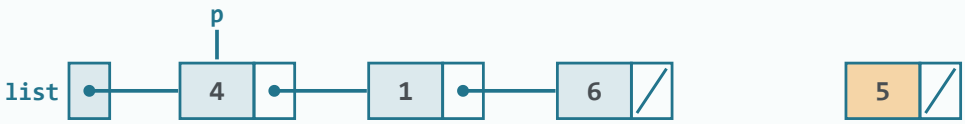
Step 1 / 4

```
1  bool insert_back(node_ptr *head, int x) {
2      node_ptr p, temp = malloc(sizeof *temp);
3      if (temp == NULL) return false;
▶      temp->data = x;
▶      temp->link = NULL;              // the new last node
6      if (*head == NULL) { *head = temp; return true; }
7      for (p = *head; p->link != NULL; p = p->link)
8          ;                           // find the last node
9      p->link = temp;
10     return true;
11 }
```

VARIABLES

`temp = &node(5)` `p = ?`

Inserting at the Back: Finding the Last Node Is O(n)



The list is not empty: `p` starts at the head.

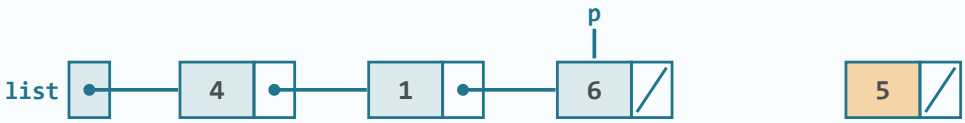
Step 2 / 4

```
1  bool insert_back(node_ptr *head, int x) {
2      node_ptr p, temp = malloc(sizeof *temp);
3      if (temp == NULL) return false;
4      temp->data = x;
5      temp->link = NULL;              // the new last node
6      if (*head == NULL) { *head = temp; return true; }
▶      for (p = *head; p->link != NULL; p = p->link)
8          ;                           // find the last node
9      p->link = temp;
10     return true;
11 }
```

VARIABLES

`temp = &node(5)` `p = &node(4)`

Inserting at the Back: Finding the Last Node Is O(n)



It advances until `p→link == NULL`: the last node is 6. This walk is **O(n)**.

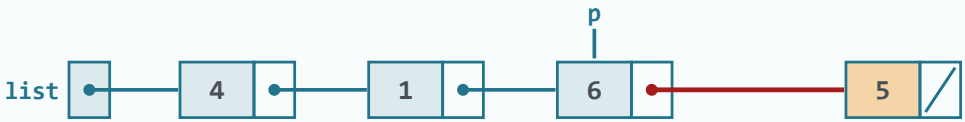
Step 3 / 4

```
1  bool insert_back(node_ptr *head, int x) {
2      node_ptr p, temp = malloc(sizeof *temp);
3      if (temp == NULL) return false;
4      temp->data = x;
5      temp->link = NULL;              // the new last node
6      if (*head == NULL) { *head = temp; return true; }
▶      for (p = *head; p->link != NULL; p = p->link)
8          ;                           // find the last node
9      p->link = temp;
10     return true;
11 }
```

VARIABLES

`temp = &node(5)` `p = &node(6)`

Inserting at the Back: Finding the Last Node Is O(n)



`p->link = temp`. If we always append, keeping a `tail` pointer makes it O(1) (that is how a linked queue works).

Step 4 / 4

```
1  bool insert_back(node_ptr *head, int x) {
2      node_ptr p, temp = malloc(sizeof *temp);
3      if (temp == NULL) return false;
4      temp->data = x;
5      temp->link = NULL;              // the new last node
6      if (*head == NULL) { *head = temp; return true; }
7      for (p = *head; p->link != NULL; p = p->link)
8          ;                           // find the last node
▶      p->link = temp;
10     return true;
11 }
```

VARIABLES

`temp = &node(5)` `p = &node(6)`

Inserting in the middle: Keep the Successor First, Then Link



12 → 7 → 25; `prev` = node(7). Insert 9 after 7.

Step 1 / 4

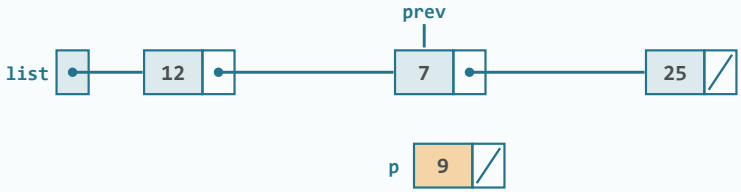
```
1 // adds a node holding x after prev
▶ node_ptr insert_after(node_ptr prev, int x) {
3     node_ptr p = new_node(x);  // malloc + check
4     if (prev != NULL) {        // middle or end
5         p->link = prev->link;
6         prev->link = p;
7     }                          // else alone
8     return p;
9 }
```

VARIABLES

`prev = &node(7)` `x = 9` `p = ?`

Generalizes `new_term(expon, coef, prev)` from polynomial multiplication.

Inserting in the middle: Keep the Successor First, Then Link



`new_node(9)` : node allocated, `link = NULL` .

Step 2 / 4

```
1 // adds a node holding x after prev
2 node_ptr insert_after(node_ptr prev, int x) {
▶     node_ptr p = new_node(x);  // malloc + check
4     if (prev != NULL) {        // middle or end
5         p->link = prev->link;
6         prev->link = p;
7     }                          // else alone
8     return p;
9 }
```

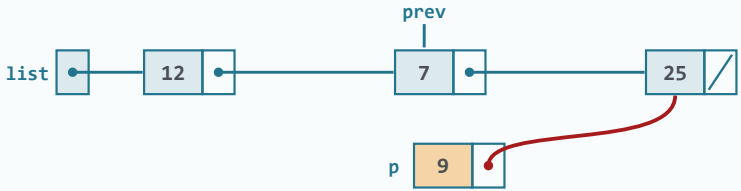
VARIABLES

`prev = &node(7)` `x = 9`

`p = &node(9)`

Generalizes `new_term(expon, coef, prev)` from polynomial multiplication.

Inserting in the middle: Keep the Successor First, Then Link



`p->link = prev->link` : 25's address is copied into the new node; nothing lost.

Step 3 / 4

```
1 // adds a node holding x after prev
2 node_ptr insert_after(node_ptr prev, int x) {
3     node_ptr p = new_node(x);  // malloc + check
4     if (prev != NULL) {        // middle or end
▶         p->link = prev->link;
6         prev->link = p;
7     }                          // else alone
8     return p;
9 }
```

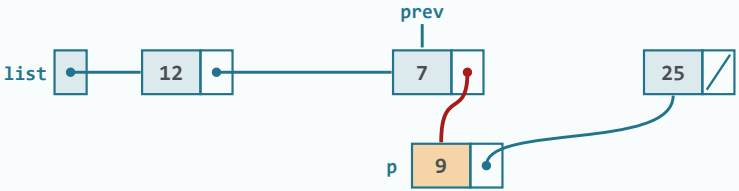
VARIABLES

`prev = &node(7)` `x = 9`

`p = &node(9)`

Generalizes `new_term(expon, coef, prev)` from polynomial multiplication.

Inserting in the middle: Keep the Successor First, Then Link



`prev->link = p` : 12 → 7 → 9 → 25. **O(1)** when the node is known.

Step 4 / 4

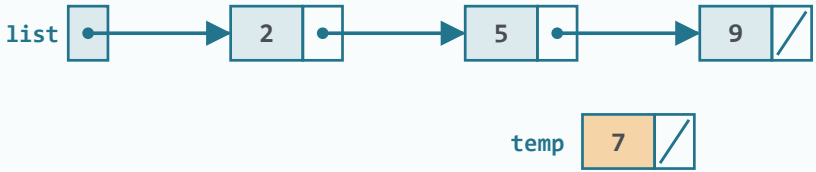
```
1 // adds a node holding x after prev
2 node_ptr insert_after(node_ptr prev, int x) {
3     node_ptr p = new_node(x);  // malloc + check
4     if (prev != NULL) {        // middle or end
5         p->link = prev->link;
▶         prev->link = p;
7     }                          // else alone
8     return p;
9 }
```

VARIABLES

`prev = &node(7)` `x = 9` `p = &node(9)`

Generalizes `new_term(expon, coef, prev)` from polynomial multiplication.

Code Trace: `insert_sorted()` — Insert 7



We insert 7 into the sorted list 2 → 5 → 9. The new node (from malloc) is ready; `temp→link` is not linked yet.

Step 1 / 12

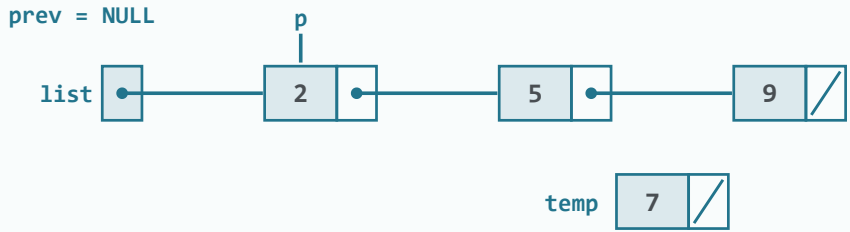
```
▶ void insert_sorted(node_ptr *head, node_ptr temp) {
2     node_ptr prev = NULL, p = *head;
3     while (p != NULL && p->data < temp->data) {
4         prev = p;
5         p = p->link;
6     }
7     temp->link = p;
8     if (prev == NULL) *head = temp;  // at the front
9     else prev->link = temp;          // middle/end
10 }
```

VARIABLES

`temp = &node(7)` `prev = ?`

`p = ?` `*head = &node(2)`

Code Trace: `insert_sorted()` — Insert 7



`p` starts at the head; `prev` will trail one step behind p (NULL for now).

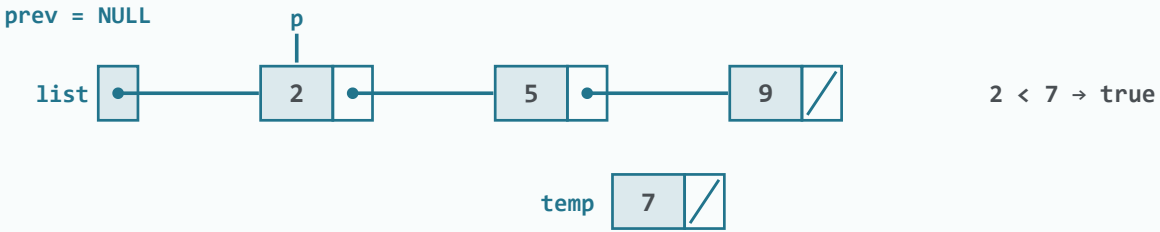
Step 2 / 12

```
1  void insert_sorted(node_ptr *head, node_ptr temp) {
▶      node_ptr prev = NULL, p = *head;
3      while (p != NULL && p->data < temp->data) {
4          prev = p;
5          p = p->link;
6      }
7      temp->link = p;
8      if (prev == NULL) *head = temp;  // at the front
9      else prev->link = temp;          // middle/end
10 }
```

VARIABLES

`temp = &node(7)` `prev = NULL` `p = &node(2)` `*head = &node(2)`

Code Trace: `insert_sorted()` — Insert 7



Condition: `2 < 7 → true`. We must go further.

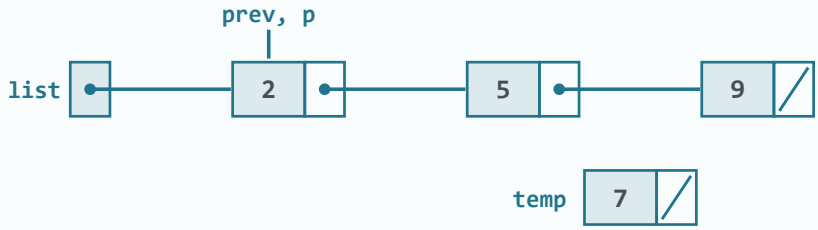
Step 3 / 12

```
1  void insert_sorted(node_ptr *head, node_ptr temp) {
2      node_ptr prev = NULL, p = *head;
▶      while (p != NULL && p->data < temp->data) {
4          prev = p;
5          p = p->link;
6      }
7      temp->link = p;
8      if (prev == NULL) *head = temp;  // at the front
9      else prev->link = temp;          // middle/end
10 }
```

VARIABLES

`temp = &node(7)` `prev = NULL`
`p = &node(2)` `*head = &node(2)`

Code Trace: `insert_sorted()` — Insert 7



`prev = p` → node(2).

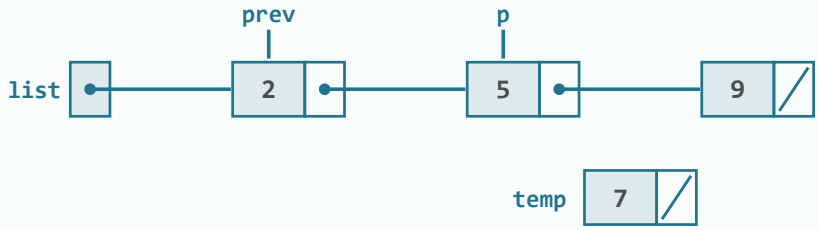
Step 4 / 12

```
1  void insert_sorted(node_ptr *head, node_ptr temp) {
2      node_ptr prev = NULL, p = *head;
3      while (p != NULL && p->data < temp->data) {
           prev = p;
5          p = p->link;
6      }
7      temp->link = p;
8      if (prev == NULL) *head = temp;  // at the front
9      else prev->link = temp;          // middle/end
10 }
```

VARIABLES

- `temp` = **&node(7)**
- `prev` = **&node(2)**
- `p` = **&node(2)**
- `*head` = **&node(2)**

Code Trace: `insert_sorted()` — Insert 7



`p = p->link` → &node(5).

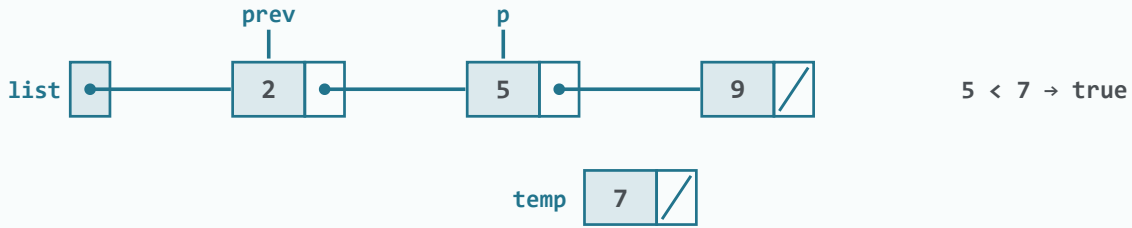
Step 5 / 12

```
1  void insert_sorted(node_ptr *head, node_ptr temp) {
2      node_ptr prev = NULL, p = *head;
3      while (p != NULL && p->data < temp->data) {
4          prev = p;
▶          p = p->link;
6      }
7      temp->link = p;
8      if (prev == NULL) *head = temp;  // at the front
9      else prev->link = temp;          // middle/end
10 }
```

VARIABLES

- `temp` = **&node(7)**
- `prev` = **&node(2)**
- `p` = **&node(5)**
- `*head` = **&node(2)**

Code Trace: `insert_sorted()` — Insert 7



Condition: `5 < 7 → true`. We must go further.

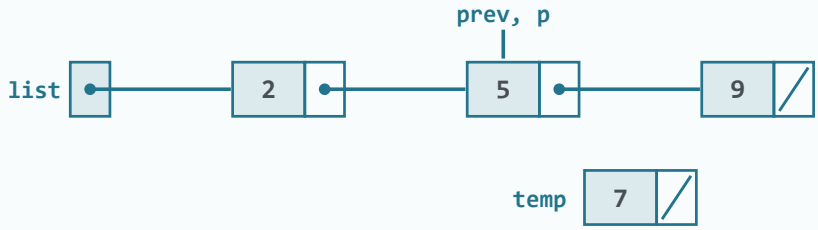
Step 6 / 12

```
1  void insert_sorted(node_ptr *head, node_ptr temp) {
2      node_ptr prev = NULL, p = *head;
▶      while (p != NULL && p->data < temp->data) {
4          prev = p;
5          p = p->link;
6      }
7      temp->link = p;
8      if (prev == NULL) *head = temp;  // at the front
9      else prev->link = temp;          // middle/end
10 }
```

VARIABLES

`temp = &node(7)` `prev = &node(2)` `p = &node(5)` `*head = &node(2)`

Code Trace: `insert_sorted()` — Insert 7



`prev = p` → node(5).

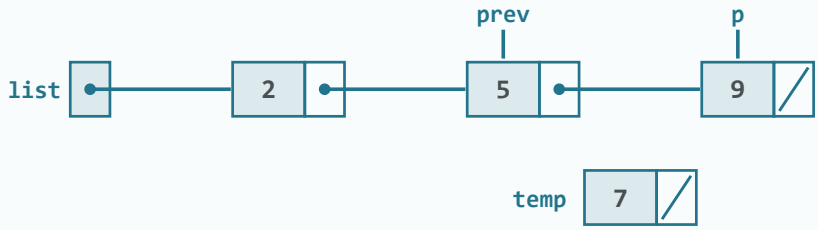
Step 7 / 12

```
1  void insert_sorted(node_ptr *head, node_ptr temp) {
2      node_ptr prev = NULL, p = *head;
3      while (p != NULL && p->data < temp->data) {
           prev = p;
5          p = p->link;
6      }
7      temp->link = p;
8      if (prev == NULL) *head = temp;  // at the front
9      else prev->link = temp;          // middle/end
10 }
```

VARIABLES

- `temp = &node(7)`
- `prev = &node(5)`
- `p = &node(5)`
- `*head = &node(2)`

Code Trace: `insert_sorted()` — Insert 7



`p = p->link` → &node(9).

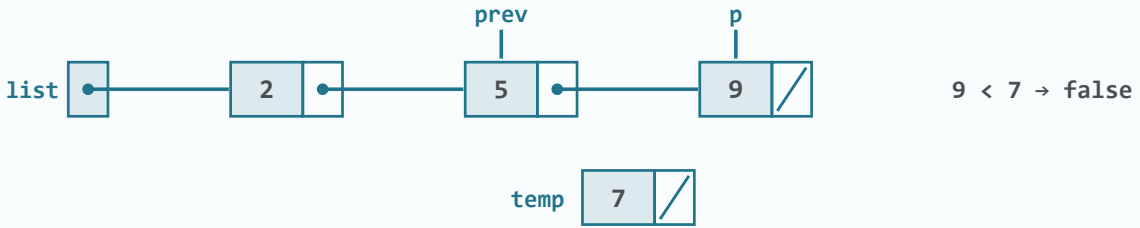
Step 8 / 12

```
1  void insert_sorted(node_ptr *head, node_ptr temp) {
2      node_ptr prev = NULL, p = *head;
3      while (p != NULL && p->data < temp->data) {
4          prev = p;
▶          p = p->link;
6      }
7      temp->link = p;
8      if (prev == NULL) *head = temp;  // at the front
9      else prev->link = temp;          // middle/end
10 }
```

VARIABLES

`temp = &node(7)` `prev = &node(5)` `p = &node(9)` `*head = &node(2)`

Code Trace: `insert_sorted()` — Insert 7



Condition: `9 < 7 → false`. The insertion point is found: **between prev and p**.

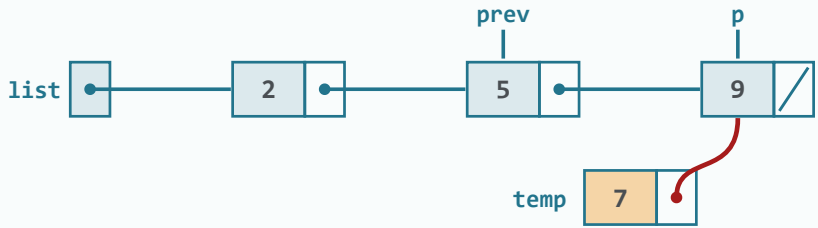
Step 9 / 12

```
1  void insert_sorted(node_ptr *head, node_ptr temp) {
2      node_ptr prev = NULL, p = *head;
▶      while (p != NULL && p->data < temp->data) {
4          prev = p;
5          p = p->link;
6      }
7      temp->link = p;
8      if (prev == NULL) *head = temp;  // at the front
9      else prev->link = temp;          // middle/end
10 }
```

VARIABLES

`temp = &node(7)` `prev = &node(5)` `p = &node(9)` `*head = &node(2)`

Code Trace: `insert_sorted()` — Insert 7



First link the new node to p: `temp→link = p` (so the address of 9 is not lost).

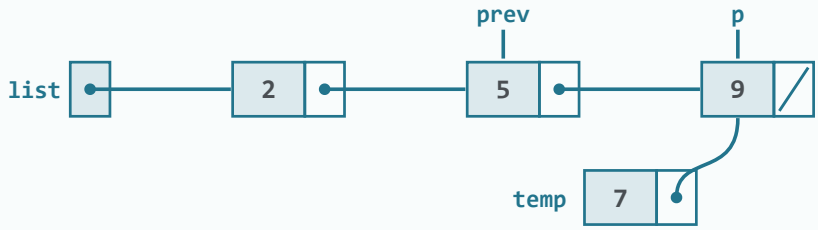
Step 10 / 12

```
1  void insert_sorted(node_ptr *head, node_ptr temp) {
2      node_ptr prev = NULL, p = *head;
3      while (p != NULL && p->data < temp->data) {
4          prev = p;
5          p = p->link;
6      }
▶      temp->link = p;
8      if (prev == NULL) *head = temp;  // at the front
9      else prev->link = temp;          // middle/end
10 }
```

VARIABLES

`temp = &node(7)` `prev = &node(5)` `p = &node(9)` `*head = &node(2)`

Code Trace: `insert_sorted()` — Insert 7



`prev ≠ NULL` : we insert in the middle, not at the front; the head stays.

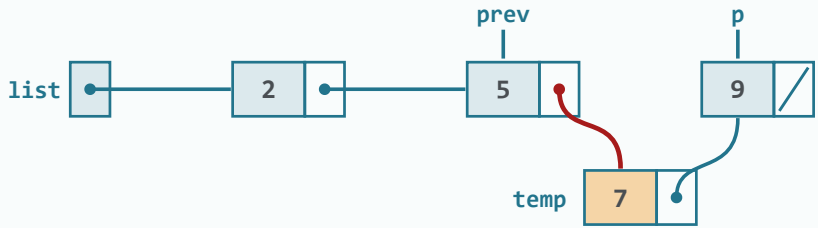
Step 11 / 12

```
1  void insert_sorted(node_ptr *head, node_ptr temp) {
2      node_ptr prev = NULL, p = *head;
3      while (p != NULL && p->data < temp->data) {
4          prev = p;
5          p = p->link;
6      }
7      temp->link = p;
▶      if (prev == NULL) *head = temp;  // at the front
9      else prev->link = temp;          // middle/end
10 }
```

VARIABLES

`temp = &node(7)` `prev = &node(5)` `p = &node(9)` `*head = &node(2)`

Code Trace: `insert_sorted()` — Insert 7



`prev->link = temp` : 5 now points to 7. The list is 2 → 5 → 7 → 9.

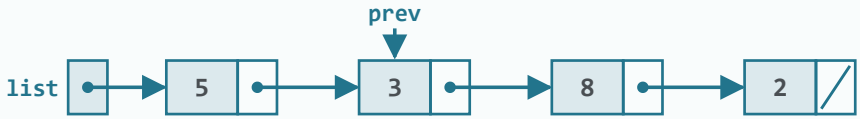
Step 12 / 12

```
1  void insert_sorted(node_ptr *head, node_ptr temp) {
2      node_ptr prev = NULL, p = *head;
3      while (p != NULL && p->data < temp->data) {
4          prev = p;
5          p = p->link;
6      }
7      temp->link = p;
8      if (prev == NULL) *head = temp;  // at the front
▶      else prev->link = temp;          // middle/end
10 }
```

VARIABLES

- `temp` = **&node(7)**
- `prev` = **&node(5)**
- `p` = **&node(9)**
- `*head` = **&node(2)**

Deletion: We Need the Node Before the Victim



To delete a node from a singly linked list we need the node **before** it: here 3.

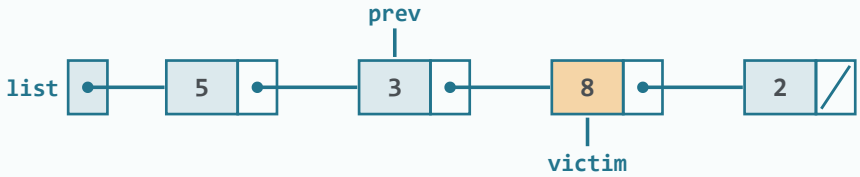
Step 1 / 4

```
1 // deletes the node after prev from the list
2 // and from memory
▶ void delete_next(node_ptr prev) {
4     node_ptr victim = prev->link;  // keep the address
5     if (victim == NULL) return;    // nothing to delete
6     prev->link = victim->link;     // fix the link
7     free(victim);                  // free the leftover
8 }
```

VARIABLES

`prev = &node(3)` `victim = ?`

Deletion: We Need the Node Before the Victim



`victim = prev→link` : keep the address, or we could not free it.

Step 2 / 4

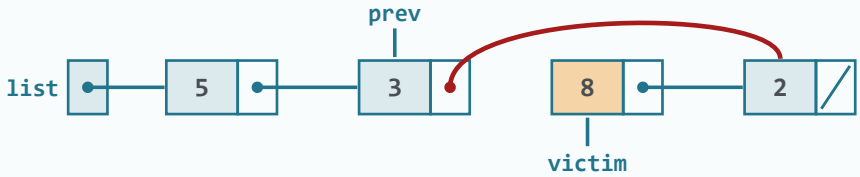
```
1 // deletes the node after prev from the list
2 // and from memory
3 void delete_next(node_ptr prev) {
▶     node_ptr victim = prev->link;  // keep the address
5     if (victim == NULL) return;    // nothing to delete
6     prev->link = victim->link;     // fix the link
7     free(victim);                  // free the leftover
8 }
```

VARIABLES

`prev = &node(3)`

`victim = &node(8)`

Deletion: We Need the Node Before the Victim



`prev->link = victim->link` : 3 now points to 2, and 8 is out of the chain.

Step 3 / 4

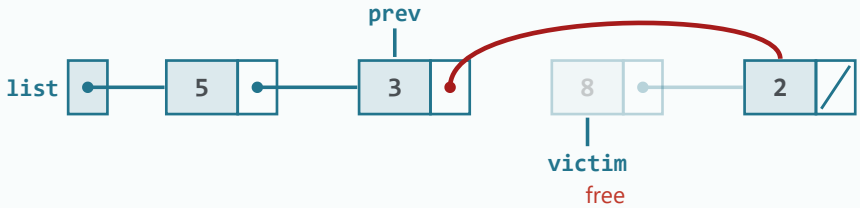
```
1 // deletes the node after prev from the list
2 // and from memory
3 void delete_next(node_ptr prev) {
4     node_ptr victim = prev->link;  // keep the address
5     if (victim == NULL) return;    // nothing to delete
▶     prev->link = victim->link;     // fix the link
7     free(victim);                  // free the leftover
8 }
```

VARIABLES

`prev = &node(3)`

`victim = &node(8)`

Deletion: We Need the Node Before the Victim



`free(victim)` : node 8 is released. Deleting the head has no predecessor → a special case (or a header node).

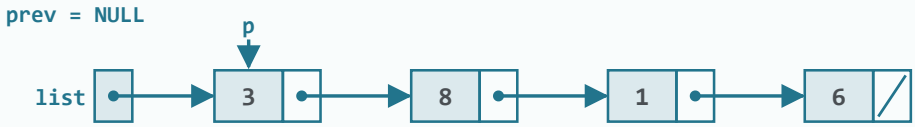
Step 4 / 4

```
1 // deletes the node after prev from the list
2 // and from memory
3 void delete_next(node_ptr prev) {
4     node_ptr victim = prev->link;  // keep the address
5     if (victim == NULL) return;    // nothing to delete
6     prev->link = victim->link;     // fix the link
▶     free(victim);                  // free the leftover
8 }
```

VARIABLES

`prev = &node(3)` `victim = (freed)`

Code Trace: `delete_key(&list, 1)`



The list is 3 → 8 → 1 → 6; we delete `x = 1`. `p` starts at the head, `prev = NULL`.

Step 1 / 13

```
1  bool delete_key(node_ptr *head, int x) {
▶      node_ptr prev = NULL, p = *head;
3      while (p != NULL && p->data != x) {
4          prev = p;
5          p = p->link;
6      }
7      if (p == NULL) return false;          // x absent
8      if (prev == NULL) *head = p->link;  // the head
9      else prev->link = p->link;            // bypass
10     free(p);
11     return true;
12 }
```

VARIABLES

`x = 1` `prev = NULL` `p = &node(3)`

Code Trace: `delete_key(&list, 1)`



Condition `3 ≠ 1 → true`. Keep searching.

Step 2 / 13

```
1  bool delete_key(node_ptr *head, int x) {
2      node_ptr prev = NULL, p = *head;
▶      while (p != NULL && p->data != x) {
4          prev = p;
5          p = p->link;
6      }
7      if (p == NULL) return false;          // x absent
8      if (prev == NULL) *head = p->link;  // the head
9      else prev->link = p->link;            // bypass
10     free(p);
11     return true;
12 }
```

VARIABLES

`x = 1` `prev = NULL` `p = &node(3)`

Code Trace: `delete_key(&list, 1)`



`prev = p` → node(3).

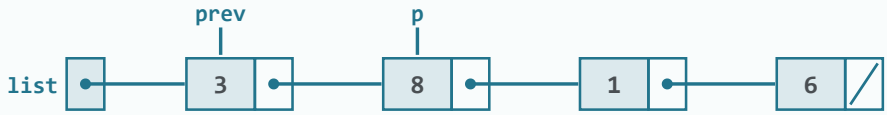
Step 3 / 13

```
1  bool delete_key(node_ptr *head, int x) {
2      node_ptr prev = NULL, p = *head;
3      while (p != NULL && p->data != x) {
▶          prev = p;
5          p = p->link;
6      }
7      if (p == NULL) return false;          // x absent
8      if (prev == NULL) *head = p->link;  // the head
9      else prev->link = p->link;            // bypass
10     free(p);
11     return true;
12 }
```

VARIABLES

`x = 1` `prev = &node(3)` `p = &node(3)`

Code Trace: `delete_key(&list, 1)`



`p = p->link` → &node(8).

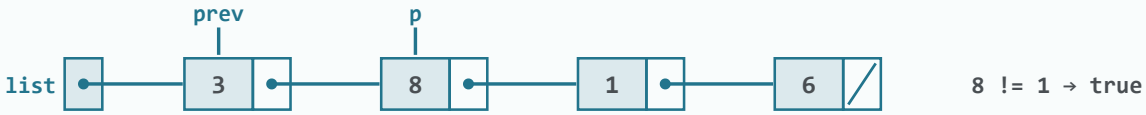
Step 4 / 13

```
1  bool delete_key(node_ptr *head, int x) {
2      node_ptr prev = NULL, p = *head;
3      while (p != NULL && p->data != x) {
4          prev = p;
▶          p = p->link;
6      }
7      if (p == NULL) return false;          // x absent
8      if (prev == NULL) *head = p->link;  // the head
9      else prev->link = p->link;            // bypass
10     free(p);
11     return true;
12 }
```

VARIABLES

`x = 1` `prev = &node(3)` `p = &node(8)`

Code Trace: `delete_key(&list, 1)`



Condition `8 ≠ 1 → true`. Keep searching.

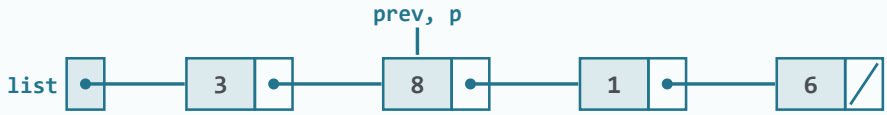
Step 5 / 13

```
1  bool delete_key(node_ptr *head, int x) {
2      node_ptr prev = NULL, p = *head;
▶      while (p != NULL && p->data != x) {
4          prev = p;
5          p = p->link;
6      }
7      if (p == NULL) return false;          // x absent
8      if (prev == NULL) *head = p->link;  // the head
9      else prev->link = p->link;            // bypass
10     free(p);
11     return true;
12 }
```

VARIABLES

- `x = 1`
- `prev = &node(3)`
- `p = &node(8)`

Code Trace: `delete_key(&list, 1)`



`prev = p` → node(8).

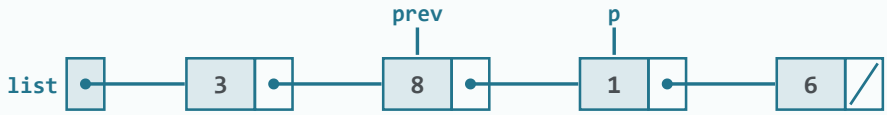
Step 6 / 13

```
1  bool delete_key(node_ptr *head, int x) {
2      node_ptr prev = NULL, p = *head;
3      while (p != NULL && p->data != x) {
▶          prev = p;
5          p = p->link;
6      }
7      if (p == NULL) return false;          // x absent
8      if (prev == NULL) *head = p->link;  // the head
9      else prev->link = p->link;            // bypass
10     free(p);
11     return true;
12 }
```

VARIABLES

`x = 1` `prev = &node(8)` `p = &node(8)`

Code Trace: `delete_key(&list, 1)`



`p = p->link` → &node(1).

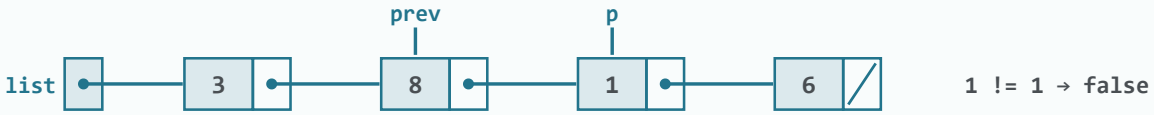
Step 7 / 13

```
1  bool delete_key(node_ptr *head, int x) {
2      node_ptr prev = NULL, p = *head;
3      while (p != NULL && p->data != x) {
4          prev = p;
▶          p = p->link;
6      }
7      if (p == NULL) return false;          // x absent
8      if (prev == NULL) *head = p->link;  // the head
9      else prev->link = p->link;            // bypass
10     free(p);
11     return true;
12 }
```

VARIABLES

`x = 1` `prev = &node(8)` `p = &node(1)`

Code Trace: `delete_key(&list, 1)`



Condition `1 ≠ 1 → false`. The node to delete is found; `prev` is one step behind.

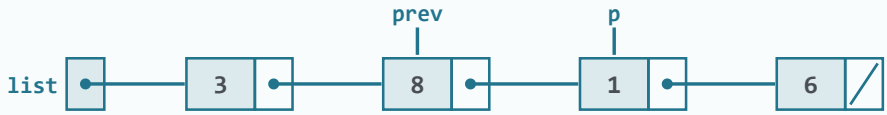
Step 8 / 13

```
1  bool delete_key(node_ptr *head, int x) {
2      node_ptr prev = NULL, p = *head;
▶      while (p != NULL && p->data != x) {
4          prev = p;
5          p = p->link;
6      }
7      if (p == NULL) return false;          // x absent
8      if (prev == NULL) *head = p->link;  // the head
9      else prev->link = p->link;            // bypass
10     free(p);
11     return true;
12 }
```

VARIABLES

`x = 1` `prev = &node(8)` `p = &node(1)`

Code Trace: `delete_key(&list, 1)`



`p ≠ NULL` : x is in the list, go on to delete it.

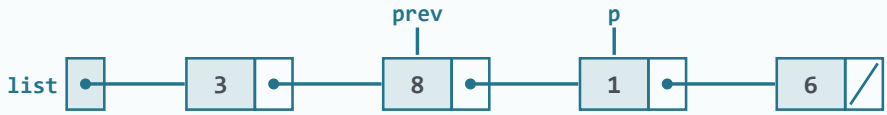
Step 9 / 13

```
1  bool delete_key(node_ptr *head, int x) {
2      node_ptr prev = NULL, p = *head;
3      while (p != NULL && p->data != x) {
4          prev = p;
5          p = p->link;
6      }
▶      if (p == NULL) return false;          // x absent
8      if (prev == NULL) *head = p->link;  // the head
9      else prev->link = p->link;           // bypass
10     free(p);
11     return true;
12 }
```

VARIABLES

`x = 1` `prev = &node(8)` `p = &node(1)`

Code Trace: `delete_key(&list, 1)`



`prev ≠ NULL` : it is not the head; `*head` will not change.

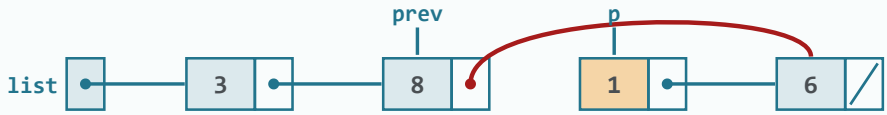
Step 10 / 13

```
1  bool delete_key(node_ptr *head, int x) {
2      node_ptr prev = NULL, p = *head;
3      while (p != NULL && p->data != x) {
4          prev = p;
5          p = p->link;
6      }
7      if (p == NULL) return false;          // x absent
▶      if (prev == NULL) *head = p->link;  // the head
9      else prev->link = p->link;            // bypass
10     free(p);
11     return true;
12 }
```

VARIABLES

`x = 1` `prev = &node(8)` `p = &node(1)`

Code Trace: `delete_key(&list, 1)`



`prev→link = p→link` : 8 now points to 6; 1 is **bypassed**.

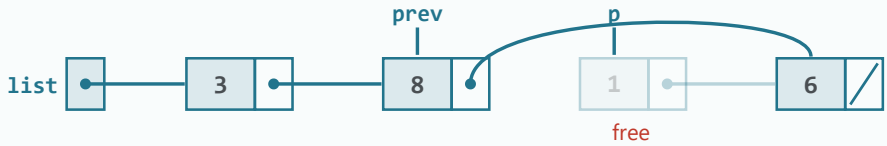
Step 11 / 13

```
1  bool delete_key(node_ptr *head, int x) {
2      node_ptr prev = NULL, p = *head;
3      while (p != NULL && p->data != x) {
4          prev = p;
5          p = p->link;
6      }
7      if (p == NULL) return false;          // x absent
8      if (prev == NULL) *head = p->link;  // the head
▶      else prev->link = p->link;           // bypass
10     free(p);
11     return true;
12 }
```

VARIABLES

`x = 1` `prev = &node(8)` `p = &node(1)`

Code Trace: `delete_key(&list, 1)`



`free(p)` : the node goes back to memory. Forgetting this causes a **memory leak**.

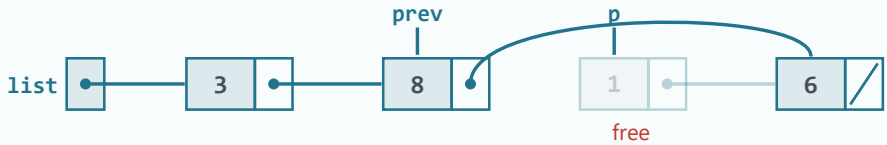
Step 12 / 13

```
1  bool delete_key(node_ptr *head, int x) {
2      node_ptr prev = NULL, p = *head;
3      while (p != NULL && p->data != x) {
4          prev = p;
5          p = p->link;
6      }
7      if (p == NULL) return false;          // x absent
8      if (prev == NULL) *head = p->link;  // the head
9      else prev->link = p->link;            // bypass
▶      free(p);
11     return true;
12 }
```

VARIABLES

`x = 1` `prev = &node(8)` `p = (freed)`

Code Trace: `delete_key(&list, 1)`



The list is 3 → 8 → 6. Search O(n), relinking O(1): **O(n)** in total.

Step 13 / 13

```
1  bool delete_key(node_ptr *head, int x) {
2      node_ptr prev = NULL, p = *head;
3      while (p != NULL && p->data != x) {
4          prev = p;
5          p = p->link;
6      }
7      if (p == NULL) return false;          // x absent
8      if (prev == NULL) *head = p->link;  // the head
9      else prev->link = p->link;            // bypass
10     free(p);
▶      return true;
12 }
```

VARIABLES

`x = 1` `prev = &node(8)` `p = (freed)`

Searching and Reaching the i-Th Node

```
node_ptr search(node_ptr p, int x) {
    while (p != NULL && p->data != x)
        p = p->link;
    return p;                // NULL if not found
}

node_ptr nth(node_ptr p, int i) {    // 0-based
    while (p != NULL && i-- > 0)
        p = p->link;
    return p;                // NULL if i >= length
}
```

Short-circuit

`p ≠ NULL && p→data ≠ x`: in the reverse order we would look inside NULL at the end of the list.

Cost

Search is O(n) in the worst case, `nth(p, i)` is O(i). Not O(1) like `a[i]` in an array.

In a sorted list

We can stop as soon as `p→data > x`; still O(n) worst case. Binary search is useless on a linked list.

Exam Question: Which One Is Right?

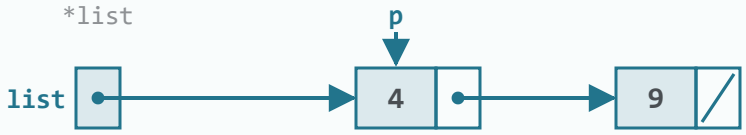
Exam question. Two routines that free a list node by node (left: a common faulty version, right: the correct one). In the left one, what happens to `list` in the caller?

```
void free_list( list_ptr *list ){
    node_ptr p = *list;
    if( *list ){
        p = *list->link;
        free(*list);
        free_list(&p);
}}
```

```
void free_list( list_ptr *list ){
    node_ptr p = *list;
    if( p ){
        *list = p->link;
        free(p);
        free_list(list);
}}
```

▶ **Solution**

Code Trace: `free_list(&lst)`



Call 1: `p = *list` → &node(4).

Step 1 / 15

```
1 void free_list(node_ptr *list) {
▶     node_ptr p = *list;
3     if (p) {
4         *list = p->link;      // first advance the
  head
5         free(p);              // then free the old
  head
6         free_list(list);      // same address again
7     }
8 }
```

VARIABLES

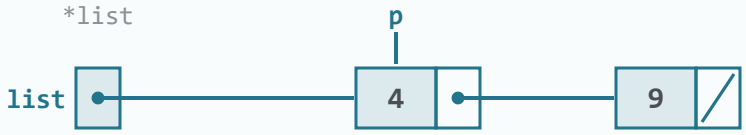
`list = &lst` `*list = &node(4)` `p = &node(4)`

CALL STACK

`free_list(&lst)`

`main()`

Code Trace: `free_list(&lst)`



`p ≠ NULL` : there is a node to free.

Step 2 / 15

```
1 void free_list(node_ptr *list) {
2     node_ptr p = *list;
▶     if (p) {
4         *list = p->link;      // first advance the
  head
5         free(p);              // then free the old
  head
6         free_list(list);      // same address again
7     }
8 }
```

VARIABLES

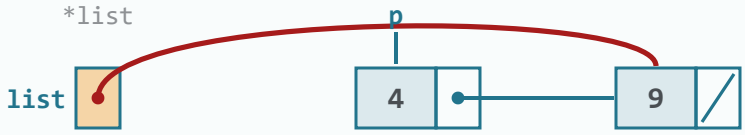
`list = &lst` `*list = &node(4)` `p = &node(4)`

CALL STACK

`free_list(&lst)`

`main()`

Code Trace: `free_list(&lst)`



`*list = p->link` : `lst` in main is now &node(9).

Step 3 / 15

```
1 void free_list(node_ptr *list) {
2     node_ptr p = *list;
3     if (p) {
▶         *list = p->link;      // first advance the head
5         free(p);              // then free the old head
6         free_list(list);      // same address again
7     }
8 }
```

VARIABLES

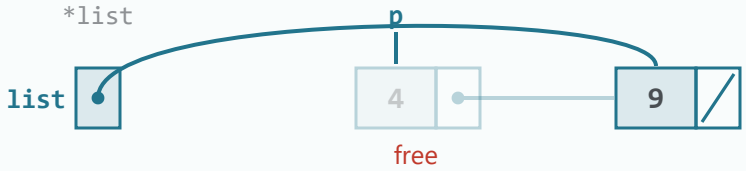
`list = &lst` `*list = &node(9)` `p = &node(4)`

CALL STACK

`free_list(&lst)`

`main()`

Code Trace: `free_list(&lst)`



`free(p)` : node(4) is released; `p` is now a dangling pointer and must not be used.

Step 4 / 15

```
1 void free_list(node_ptr *list) {
2     node_ptr p = *list;
3     if (p) {
4         *list = p->link;      // first advance the head
          free(p);              // then free the old head
6         free_list(list);      // same address again
7     }
8 }
```

VARIABLES

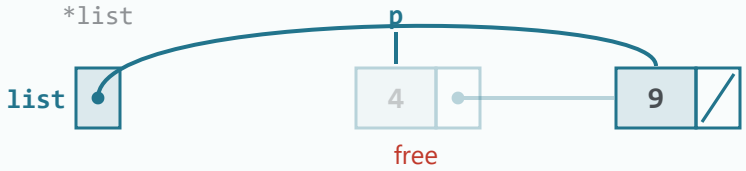
`list = &lst` `*list = &node(9)` `p = &node(4) (freed)`

CALL STACK

`free_list(&lst)`

`main()`

Code Trace: `free_list(&lst)`



Recursive call with the same `list` address: free the rest of the list.

Step 5 / 15

```
1 void free_list(node_ptr *list) {
2     node_ptr p = *list;
3     if (p) {
4         *list = p->link;      // first advance the head
5         free(p);              // then free the old head
▶         free_list(list);      // same address again
7     }
8 }
```

VARIABLES

`list = &lst` `*list = &node(9)` `p = &node(4) (freed)`

CALL STACK

- `free_list(&lst)`
- `main()`

Code Trace: `free_list(&lst)`



Call 2: `p = *list` → &node(9).

Step 6 / 15

```
1  void free_list(node_ptr *list) {
▶      node_ptr p = *list;
3      if (p) {
4          *list = p->link;      // first advance the head
5          free(p);              // then free the old head
6          free_list(list);      // same address again
7      }
8  }
```

VARIABLES

`list = &lst` `*list = &node(9)` `p = &node(9)`

CALL STACK

- `free_list(&lst)`
- `free_list(&lst)`
- `main()`

Code Trace: `free_list(&lst)`



`p ≠ NULL` : there is a node to free.

Step 7 / 15

```
1 void free_list(node_ptr *list) {
2     node_ptr p = *list;
▶     if (p) {
4         *list = p->link;      // first advance the
  head
5         free(p);              // then free the old
  head
6         free_list(list);      // same address again
7     }
8 }
```

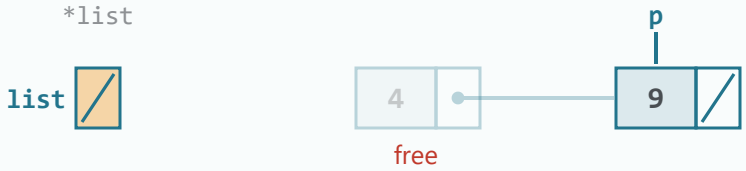
VARIABLES

`list` = **`&lst`** | `*list` = **`&node(9)`** | `p` = **`&node(9)`**

CALL STACK

- **`free_list(&lst)`**
- `free_list(&lst)`
- `main()`

Code Trace: `free_list(&lst)`



`*list = p->link` : `lst` in main is now NULL.

Step 8 / 15

```
1 void free_list(node_ptr *list) {
2     node_ptr p = *list;
3     if (p) {
▶         *list = p->link;      // first advance the head
5         free(p);              // then free the old head
6         free_list(list);      // same address again
7     }
8 }
```

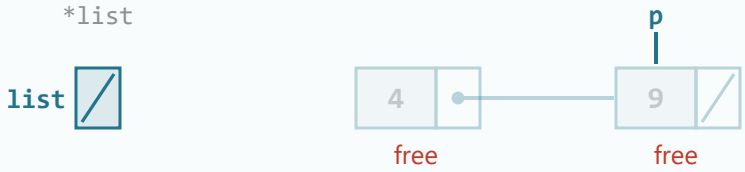
VARIABLES

`list = &lst` `*list = NULL` `p = &node(9)`

CALL STACK

- `free_list(&lst)`
- `free_list(&lst)`
- `main()`

Code Trace: `free_list(&lst)`



`free(p)` : node(9) is released; `p` is now a dangling pointer and must not be used.

Step 9 / 15

```
1 void free_list(node_ptr *list) {
2     node_ptr p = *list;
3     if (p) {
4         *list = p->link;      // first advance the head
▶         free(p);              // then free the old head
6         free_list(list);      // same address again
7     }
8 }
```

VARIABLES

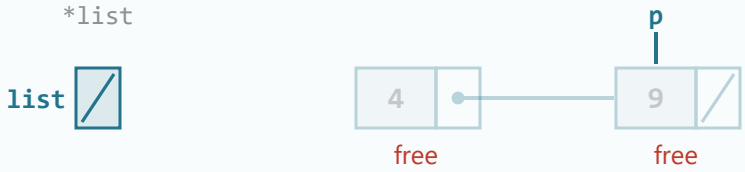
`list = &lst` `*list = NULL`

`p = &node(9) (freed)`

CALL STACK

- `free_list(&lst)`
- `free_list(&lst)`
- `main()`

Code Trace: `free_list(&lst)`



Recursive call with the same `list` address: free the rest of the list.

Step 10 / 15

```
1 void free_list(node_ptr *list) {
2     node_ptr p = *list;
3     if (p) {
4         *list = p->link;      // first advance the head
5         free(p);              // then free the old head
▶         free_list(list);      // same address again
7     }
8 }
```

VARIABLES

`list = &lst` `*list = NULL`

`p = &node(9) (freed)`

CALL STACK

- **free_list(&lst)**
- free_list(&lst)
- main()

Code Trace: `free_list(&lst)`



Call 3: `p = *list` → NULL.


```
1 void free_list(node_ptr *list) {
▶     node_ptr p = *list;
3     if (p) {
4         *list = p->link;      // first advance the
  head
5         free(p);              // then free the old
  head
6         free_list(list);      // same address again
7     }
8 }
```

VARIABLES

`list = &lst` `*list = NULL`

`p = NULL`

CALL STACK

- **free_list(&lst)**
- free_list(&lst)
- free_list(&lst)
- main()

Code Trace: `free_list(&lst)`



`p == NULL` : the list is done; the call returns without doing anything. **Base case.**

Step 12 / 15

```
1 void free_list(node_ptr *list) {
2     node_ptr p = *list;
▶     if (p) {
4         *list = p->link;      // first advance the
  head
5         free(p);              // then free the old
  head
6         free_list(list);      // same address again
7     }
8 }
```

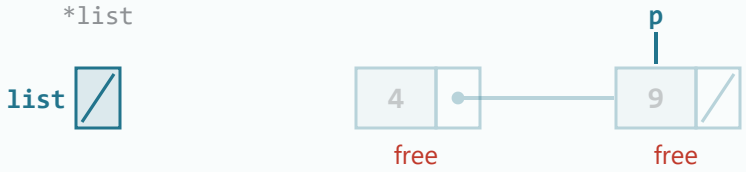
VARIABLES

`list = &lst` `*list = NULL` `p = NULL`

CALL STACK

- **`free_list(&lst)`**
- `free_list(&lst)`
- `free_list(&lst)`
- `main()`

Code Trace: `free_list(&lst)`



Back in call 2; nothing left to do.

Step 13 / 15

```
1 void free_list(node_ptr *list) {
2     node_ptr p = *list;
3     if (p) {
4         *list = p->link;      // first advance the head
5         free(p);              // then free the old head
6         free_list(list);      // same address again
▶     }
8 }
```

VARIABLES

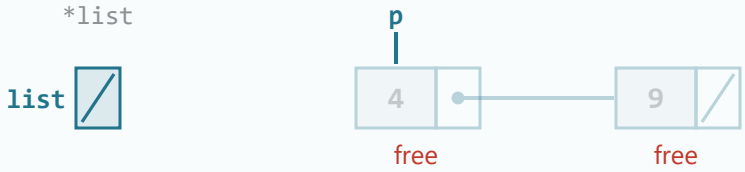
`list = &lst` `*list = NULL`

`p = &node(9) (freed)`

CALL STACK

- `free_list(&lst)`
- `free_list(&lst)`
- `main()`

Code Trace: `free_list(&lst)`



Back in call 1; nothing left to do.

Step 14 / 15

```
1 void free_list(node_ptr *list) {
2     node_ptr p = *list;
3     if (p) {
4         *list = p->link;      // first advance the
  head
5         free(p);              // then free the old
  head
6         free_list(list);      // same address again
▶     }
8 }
```

VARIABLES

`list = &lst` `*list = NULL`

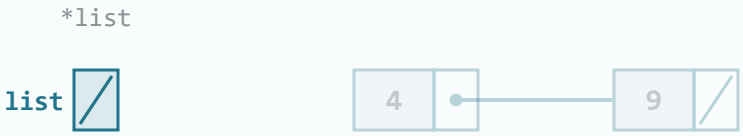
`p = &node(4) (freed)`

CALL STACK

`free_list(&lst)`

`main()`

Code Trace: `free_list(&lst)`



Every node is freed and `lst == NULL` in main: no dangling pointer is left.

Step 15 / 15

```
1 void free_list(node_ptr *list) {
2     node_ptr p = *list;
3     if (p) {
4         *list = p->link;      // first advance the
  head
5         free(p);              // then free the old
  head
6         free_list(list);      // same address again
▶     }
8 }
```

VARIABLES

`lst = NULL`

CALL STACK

`main()`

The Same Job, Iteratively

```
void clear_list(node_ptr *list) {
    while (*list != NULL) {
        node_ptr p = *list;
        *list = p->link;     // advance first
        free(p);             // then free
    }
}
```

Recursive version

One call frame per node: with n = 10^6 nodes the stack may overflow.

Iterative version

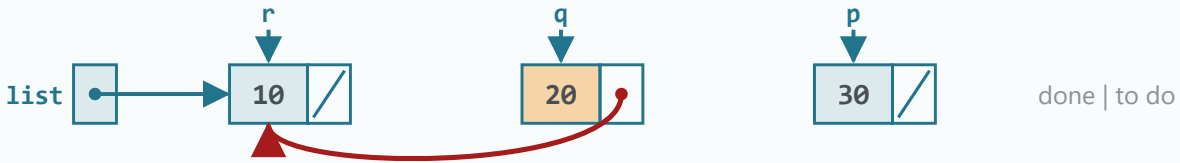
O(n) time, **O(1)** extra space; `*list` ends up NULL.

The common rule

Take the next address **before** free; never read the `link` of a freed node.

Question: Reverse the Links (`invert`)

QUESTION. Write a routine `invert` that reverses the link fields of the list pointed to by `list`.



Midway: the part up to r and q is reversed, the part from p on still points the old way.

Three pointers

`r` trails, `q` is in the middle, `p` leads; each round reverses one link.

In place

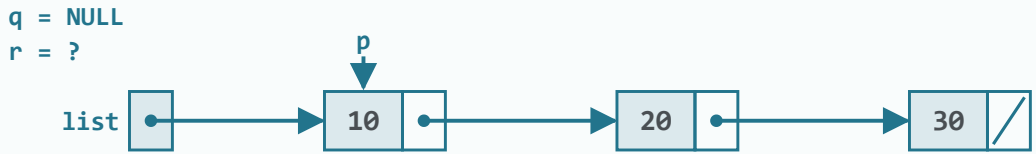
No new nodes, no copies: O(n) time, O(1) extra space.

The head changes

That is why the list is passed by address as `node_ptr *list`.

Instructor "No add_node, no remove_node, no malloc, no extra memory, no extra work; nothing at all!"

Code Trace: `invert(&list)`



Three pointers: `p` leads (the head), `q` is in the middle (NULL), `r` trails (no value yet).

Step 1 / 18

```
1  void invert(node_ptr *list) {
▶      node_ptr r, p = *list, q = NULL;
3      while (p) {
4          r = q;          // r trails q
5          q = p;          // q trails p
6          p = p->link;    // advance p
7          q->link = r;    // reverse the link
8      }
9      *list = q;          // last node is the head
10 }
```

VARIABLES

`r = ?` `q = NULL` `p = &node(10)` `*list = &node(10)`

Code Trace: `invert(&list)`



`p ≠ NULL` : a round begins.

Step 2 / 18

```
1  void invert(node_ptr *list) {
2      node_ptr r, p = *list, q = NULL;
▶      while (p) {
4          r = q;          // r trails q
5          q = p;          // q trails p
6          p = p->link;    // advance p
7          q->link = r;    // reverse the link
8      }
9      *list = q;          // last node is the head
10 }
```

VARIABLES

`r = ?` `q = NULL` `p = &node(10)` `*list = &node(10)`

Code Trace: `invert(&list)`



Round 1: `r = q` → NULL.

Step 3 / 18

```
1  void invert(node_ptr *list) {
2      node_ptr r, p = *list, q = NULL;
3      while (p) {
▶          r = q;          // r trails q
5          q = p;          // q trails p
6          p = p->link;    // advance p
7          q->link = r;    // reverse the link
8      }
9      *list = q;          // last node is the head
10 }
```

VARIABLES

`r = NULL` `q = NULL` `p = &node(10)` `*list = &node(10)`

Code Trace: `invert(&list)`



`q = p` → &node(10): the node whose link is reversed this round.

Step 4 / 18

```
1  void invert(node_ptr *list) {
2      node_ptr r, p = *list, q = NULL;
3      while (p) {
4          r = q;          // r trails q
▶          q = p;          // q trails p
6          p = p->link;    // advance p
7          q->link = r;    // reverse the link
8      }
9      *list = q;          // last node is the head
10 }
```

VARIABLES

`r = NULL` `q = &node(10)` `p = &node(10)` `*list = &node(10)`

Code Trace: `invert(&list)`



`p = p→link` → &node(20). We advance **first** so the rest of the list is not lost.

Step 5 / 18

```
1  void invert(node_ptr *list) {
2      node_ptr r, p = *list, q = NULL;
3      while (p) {
4          r = q;          // r trails q
5          q = p;          // q trails p
▶          p = p->link;    // advance p
7          q->link = r;    // reverse the link
8      }
9      *list = q;          // last node is the head
10 }
```

VARIABLES

`r = NULL` `q = &node(10)` `p = &node(20)` `*list = &node(10)`

Code Trace: `invert(&list)`



`q→link = r` : node(10) now points to NULL.

Step 6 / 18

```
1  void invert(node_ptr *list) {
2      node_ptr r, p = *list, q = NULL;
3      while (p) {
4          r = q;          // r trails q
5          q = p;          // q trails p
6          p = p->link;    // advance p
▶          q->link = r;    // reverse the link
8      }
9      *list = q;          // last node is the head
10 }
```

VARIABLES

`r = NULL` `q = &node(10)` `p = &node(20)` `*list = &node(10)`

Code Trace: `invert(&list)`



`p ≠ NULL` : a round begins.

Step 7 / 18

```
1  void invert(node_ptr *list) {
2      node_ptr r, p = *list, q = NULL;
▶      while (p) {
4          r = q;          // r trails q
5          q = p;          // q trails p
6          p = p->link;    // advance p
7          q->link = r;    // reverse the link
8      }
9      *list = q;          // last node is the head
10 }
```

VARIABLES

`r = NULL` `q = &node(10)` `p = &node(20)` `*list = &node(10)`

Code Trace: `invert(&list)`



Round 2: `r = q` → &node(10).

Step 8 / 18

```
1  void invert(node_ptr *list) {
2      node_ptr r, p = *list, q = NULL;
3      while (p) {
▶          r = q;          // r trails q
5          q = p;          // q trails p
6          p = p->link;    // advance p
7          q->link = r;    // reverse the link
8      }
9      *list = q;          // last node is the head
10 }
```

VARIABLES

`r = &node(10)` `q = &node(10)` `p = &node(20)` `*list = &node(10)`

Code Trace: `invert(&list)`



`q = p` → &node(20): the node whose link is reversed this round.

Step 9 / 18

```
1  void invert(node_ptr *list) {
2      node_ptr r, p = *list, q = NULL;
3      while (p) {
4          r = q;          // r trails q
▶          q = p;          // q trails p
6          p = p->link;    // advance p
7          q->link = r;    // reverse the link
8      }
9      *list = q;          // last node is the head
10 }
```

VARIABLES

r = &node(10)	q = &node(20)
p = &node(20)	*list = &node(10)

Code Trace: `invert(&list)`



`p = p⟶link` → &node(30). We advance **first** so the rest of the list is not lost.

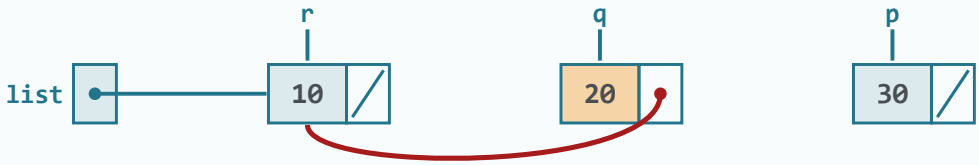
Step 10 / 18

```
1  void invert(node_ptr *list) {
2      node_ptr r, p = *list, q = NULL;
3      while (p) {
4          r = q;          // r trails q
5          q = p;          // q trails p
▶          p = p->link;    // advance p
7          q->link = r;    // reverse the link
8      }
9      *list = q;          // last node is the head
10 }
```

VARIABLES

r = &node(10)	q = &node(20)
p = &node(30)	*list = &node(10)

Code Trace: `invert(&list)`



`q→link = r` : node(20) now points to node(10).

Step 11 / 18

```
1  void invert(node_ptr *list) {
2      node_ptr r, p = *list, q = NULL;
3      while (p) {
4          r = q;          // r trails q
5          q = p;          // q trails p
6          p = p->link;    // advance p
▶          q->link = r;    // reverse the link
8      }
9      *list = q;          // last node is the head
10 }
```

VARIABLES

`r = &node(10)` `q = &node(20)` `p = &node(30)` `*list = &node(10)`

Code Trace: `invert(&list)`



`p ≠ NULL` : a round begins.

Step 12 / 18

```
1  void invert(node_ptr *list) {
2      node_ptr r, p = *list, q = NULL;
▶      while (p) {
4          r = q;          // r trails q
5          q = p;          // q trails p
6          p = p->link;    // advance p
7          q->link = r;    // reverse the link
8      }
9      *list = q;          // last node is the head
10 }
```

VARIABLES

- `r = &node(10)`
- `q = &node(20)`
- `p = &node(30)`
- `*list = &node(10)`

Code Trace: `invert(&list)`



Round 3: `r = q` → &node(20).

Step 13 / 18

```
1  void invert(node_ptr *list) {
2      node_ptr r, p = *list, q = NULL;
3      while (p) {
▶          r = q;          // r trails q
5          q = p;          // q trails p
6          p = p->link;    // advance p
7          q->link = r;    // reverse the link
8      }
9      *list = q;          // last node is the head
10 }
```

VARIABLES

`r = &node(20)` `q = &node(20)` `p = &node(30)` `*list = &node(10)`

Code Trace: `invert(&list)`



`q = p` → &node(30): the node whose link is reversed this round.

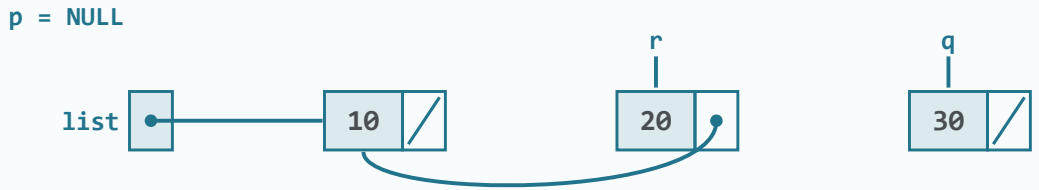
Step 14 / 18

```
1  void invert(node_ptr *list) {
2      node_ptr r, p = *list, q = NULL;
3      while (p) {
4          r = q;          // r trails q
▶          q = p;          // q trails p
6          p = p->link;    // advance p
7          q->link = r;    // reverse the link
8      }
9      *list = q;          // last node is the head
10 }
```

VARIABLES

- `r` = &node(20)
- `q` = &node(30)
- `p` = &node(30)
- `*list` = &node(10)

Code Trace: `invert(&list)`



`p = p→link` → NULL. We advance **first** so the rest of the list is not lost.

Step 15 / 18

```
1  void invert(node_ptr *list) {
2      node_ptr r, p = *list, q = NULL;
3      while (p) {
4          r = q;          // r trails q
5          q = p;          // q trails p
▶          p = p->link;    // advance p
7          q->link = r;    // reverse the link
8      }
9      *list = q;          // last node is the head
10 }
```

VARIABLES

`r = &node(20)` `q = &node(30)` `p = NULL` `*list = &node(10)`

Code Trace: `invert(&list)`



`q→link = r` : node(30) now points to node(20).

Step 16 / 18

```
1  void invert(node_ptr *list) {
2      node_ptr r, p = *list, q = NULL;
3      while (p) {
4          r = q;          // r trails q
5          q = p;          // q trails p
6          p = p->link;    // advance p
▶          q->link = r;    // reverse the link
8      }
9      *list = q;          // last node is the head
10 }
```

VARIABLES

`r = &node(20)` `q = &node(30)` `p = NULL` `*list = &node(10)`

Code Trace: `invert(&list)`



`p == NULL` : the loop ends; q is at the last node.

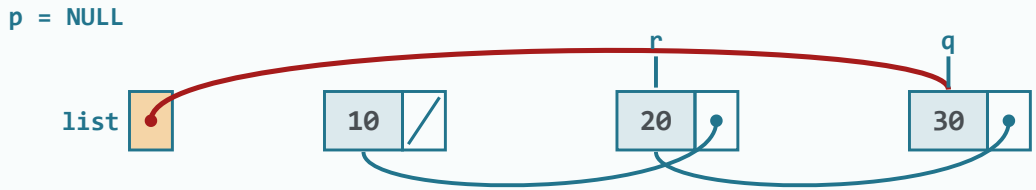
Step 17 / 18

```
1  void invert(node_ptr *list) {
2      node_ptr r, p = *list, q = NULL;
▶      while (p) {
4          r = q;          // r trails q
5          q = p;          // q trails p
6          p = p->link;    // advance p
7          q->link = r;    // reverse the link
8      }
9      *list = q;          // last node is the head
10 }
```

VARIABLES

`r = &node(20)` `q = &node(30)` `p = NULL` `*list = &node(10)`

Code Trace: `invert(&list)`



`*list = q` : the old last node is the new head. 30 → 20 → 10; **O(n)** time, **O(1)** extra space, no malloc.

Step 18 / 18

```
1  void invert(node_ptr *list) {
2      node_ptr r, p = *list, q = NULL;
3      while (p) {
4          r = q;          // r trails q
5          q = p;          // q trails p
6          p = p->link;    // advance p
7          q->link = r;    // reverse the link
8      }
▶      *list = q;          // last node is the head
10 }
```

VARIABLES

`r = &node(20)` `q = &node(30)` `p = NULL` `*list = &node(30)`

Concatenating Two Lists End to End



L1 = 1 → 4, L2 = 6 → 9. Neither is empty.

Step 1 / 4

```
1  // links L2 to the end of L1, returns the head
2  node_ptr concat(node_ptr L1, node_ptr L2) {
3      node_ptr p;
▶      if (L1 == NULL) return L2;
▶      if (L2 != NULL) {
6          for (p = L1; p->link != NULL; p = p->link)
7              ;                    // last node of L1
8          p->link = L2;
9      }
10     return L1;
11 }
```

VARIABLES

L1 = &node(1) L2 = &node(6) p = ?

Concatenating Two Lists End to End



`p` walks from the head of L1.

Step 2 / 4

```
1  // links L2 to the end of L1, returns the head
2  node_ptr concat(node_ptr L1, node_ptr L2) {
3      node_ptr p;
4      if (L1 == NULL) return L2;
5      if (L2 != NULL) {
▶          for (p = L1; p->link != NULL; p = p->link)
7              ;                    // last node of L1
8          p->link = L2;
9      }
10     return L1;
11 }
```

VARIABLES

`L1 = &node(1)` `L2 = &node(6)` `p = &node(1)`

Concatenating Two Lists End to End



`p→link == NULL` : the last node of L1 is found. Cost **O(|L1|)**.

Step 3 / 4

```
1  // links L2 to the end of L1, returns the head
2  node_ptr concat(node_ptr L1, node_ptr L2) {
3      node_ptr p;
4      if (L1 == NULL) return L2;
5      if (L2 != NULL) {
▶          for (p = L1; p->link != NULL; p = p->link)
7              ;                    // last node of L1
8          p->link = L2;
9      }
10     return L1;
11 }
```

VARIABLES

L1 = &node(1) L2 = &node(6)

p = &node(4)

Concatenating Two Lists End to End



`p→link = L2` : 1 → 4 → 6 → 9. No copies. With a `last` pointer on a circular list this is O(1).

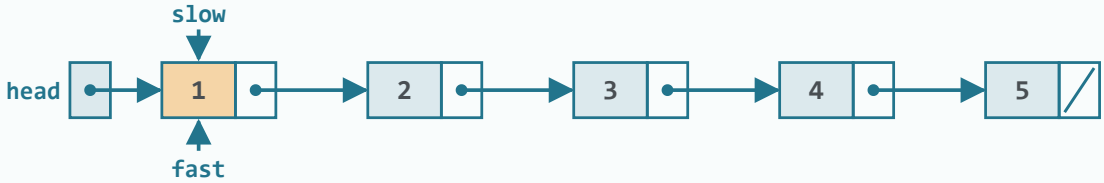
Step 4 / 4

```
1  // links L2 to the end of L1, returns the head
2  node_ptr concat(node_ptr L1, node_ptr L2) {
3      node_ptr p;
4      if (L1 == NULL) return L2;
5      if (L2 != NULL) {
6          for (p = L1; p->link != NULL; p = p->link)
7              ;                    // last node of L1
▶          p->link = L2;
9      }
10     return L1;
11 }
```

VARIABLES

`L1 = &node(1)` `L2 = &node(6)` `p = &node(4)`

The middle Node: Slow and Fast Pointers



Both start at the head.

Step 1 / 4

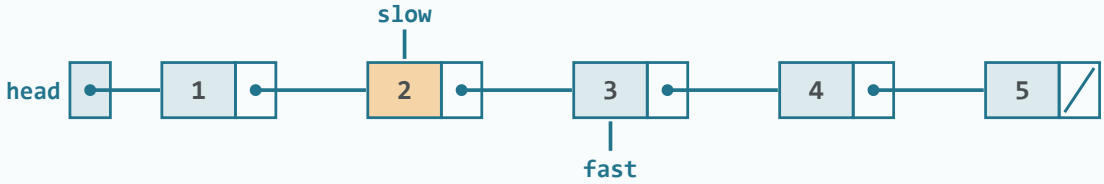
```
1 node_ptr middle(node_ptr head) {
▶     node_ptr slow = head, fast = head;
3     while (fast != NULL && fast->link != NULL) {
4         slow = slow->link;          // 1 step
5         fast = fast->link->link;    // 2 steps
6     }
7     return slow;
8 }
```

VARIABLES

`slow = &node(1)` `fast = &node(1)`

For even length (1 → 2 → 3 → 4) the second middle (3) is returned; for the first middle use the condition `fast→link→link ≠ NULL`.

The middle Node: Slow and Fast Pointers



slow takes 1 step, fast takes 2.

Step 2 / 4

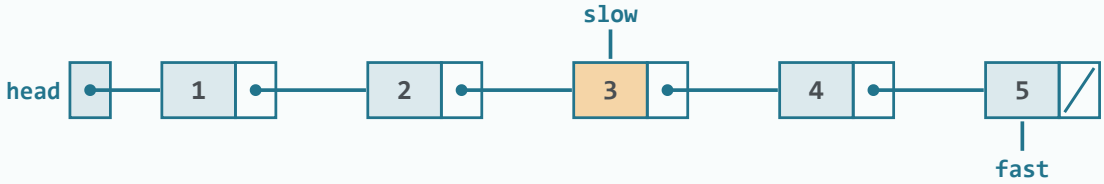
```
1 node_ptr middle(node_ptr head) {
2     node_ptr slow = head, fast = head;
3     while (fast != NULL && fast->link != NULL) {
▶         slow = slow->link;          // 1 step
▶         fast = fast->link->link;    // 2 steps
6     }
7     return slow;
8 }
```

VARIABLES

`slow = &node(2)` `fast = &node(3)`

For even length (1 → 2 → 3 → 4) the second middle (3) is returned; for the first middle use the condition `fast→link→link ≠ NULL`.

The middle Node: Slow and Fast Pointers



fast is at the last node (`fast→link == NULL`): the loop ends and **slow = 3** is in the middle.

Step 3 / 4

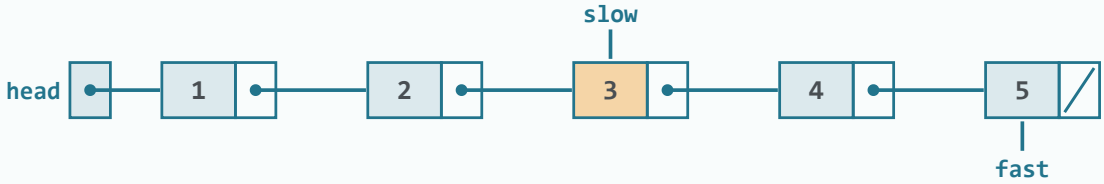
```
1 node_ptr middle(node_ptr head) {
2     node_ptr slow = head, fast = head;
3     while (fast != NULL && fast->link != NULL) {
          slow = slow->link;          // 1 step
          fast = fast->link->link;    // 2 steps
6     }
7     return slow;
8 }
```

VARIABLES

`slow = &node(3)` `fast = &node(5)`

For even length (1 → 2 → 3 → 4) the second middle (3) is returned; for the first middle use the condition `fast→link→link ≠ NULL` .

The middle Node: Slow and Fast Pointers



`return slow` : one pass, O(n) time, O(1) extra space.

Step 4 / 4

```
1 node_ptr middle(node_ptr head) {
2     node_ptr slow = head, fast = head;
3     while (fast != NULL && fast->link != NULL) {
4         slow = slow->link;          // 1 step
5         fast = fast->link->link;    // 2 steps
6     }
▶     return slow;
8 }
```

VARIABLES

`slow = &node(3)` `fast = &node(5)`

For even length (1 → 2 → 3 → 4) the second middle (3) is returned; for the first middle use the condition `fast→link→link ≠ NULL`.

CLASSIC ROUTINES

Floyd's Cycle-Detection Algorithm (Tortoise and Hare)

The problem

A faulty link (`last→link = some_middle_node`) puts the list in a cycle; `print_list` never ends.

The idea

slow takes 1 step, fast takes 2. Without a cycle fast reaches NULL; with one, once both are in the cycle the gap shrinks by 1 each round → they must meet.

Cost

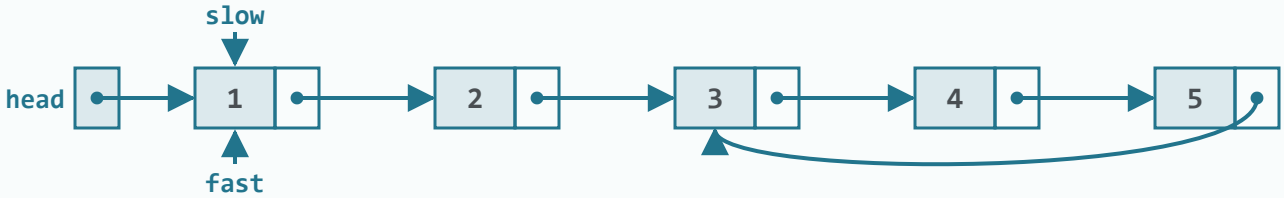
O(n) time, **O(1)** extra space. Keeping visited nodes in a set also works but needs O(n) space.

Start of the cycle

After they meet, move one pointer back to the head and advance both one step at a time: the node where they meet again is the cycle's entry.

Floyd (1967); Knuth, TAOCP Vol. 2, Exercise 3.1-6.

Code Trace: `has_cycle()` — Tortoise and Hare



5 → 3: cycle

The tortoise (`slow`) and the hare (`fast`) start together at the head. The list has a cycle 5 → 3.

Step 1 / 13

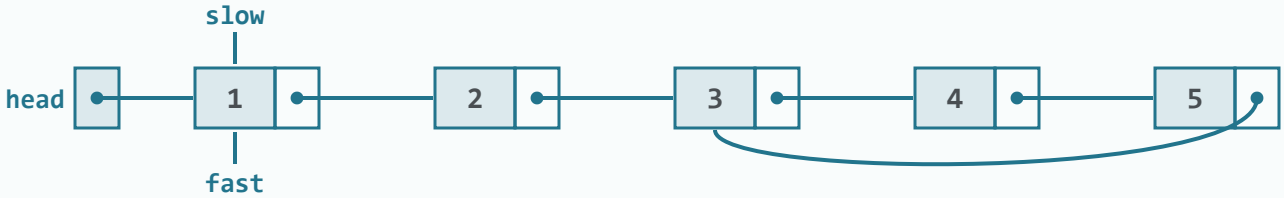
```
1 bool has_cycle(node_ptr head) {
▶       node_ptr slow = head, fast = head;
3       while (fast != NULL && fast->link != NULL) {
4           slow = slow->link;          // 1 step
5           fast = fast->link->link;    // 2 steps
6           if (slow == fast) return true;
7       }
8       return false;
9   }
```

VARIABLES

`slow` = **`&node(1)`**

`fast` = **`&node(1)`**

Code Trace: `has_cycle()` — Tortoise and Hare



5 → 3: cycle

`fast` and its successor are not NULL: keep running.

Step 2 / 13

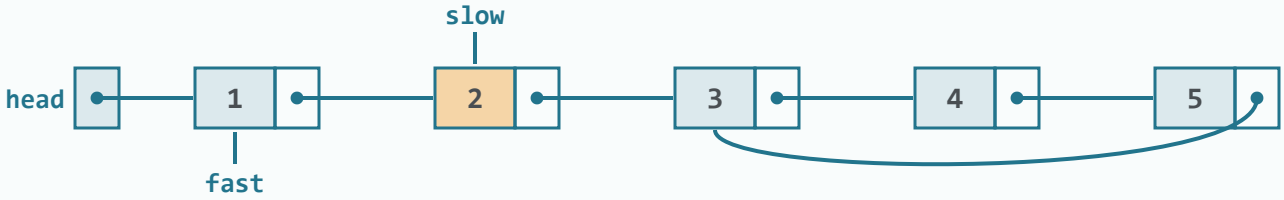
```
1 bool has_cycle(node_ptr head) {
2     node_ptr slow = head, fast = head;
▶     while (fast != NULL && fast->link != NULL) {
4         slow = slow->link;          // 1 step
5         fast = fast->link->link;    // 2 steps
6         if (slow == fast) return true;
7     }
8     return false;
9 }
```

VARIABLES

`slow` = **&node(1)**

`fast` = **&node(1)**

Code Trace: `has_cycle()` — Tortoise and Hare



5 → 3: cycle

Round 1: the tortoise takes 1 step → node(2).

Step 3 / 13

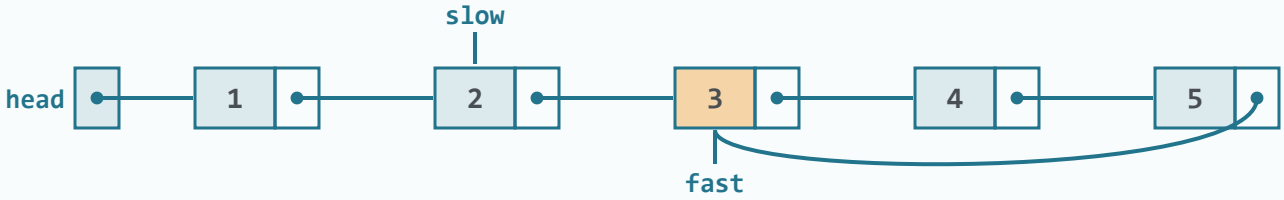
```
1 bool has_cycle(node_ptr head) {
2     node_ptr slow = head, fast = head;
3     while (fast != NULL && fast->link != NULL) {
▶         slow = slow->link;          // 1 step
5         fast = fast->link->link;    // 2 steps
6         if (slow == fast) return true;
7     }
8     return false;
9 }
```

VARIABLES

`slow = &node(2)`

`fast = &node(1)`

Code Trace: `has_cycle()` — Tortoise and Hare



5 → 3: cycle

The hare takes 2 steps → node(3).

Step 4 / 13

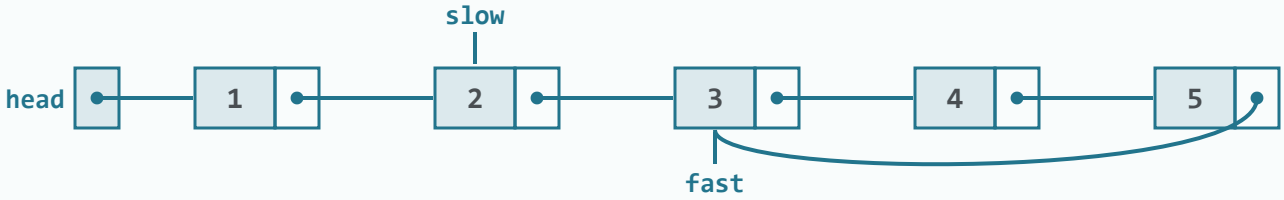
```
1 bool has_cycle(node_ptr head) {
2     node_ptr slow = head, fast = head;
3     while (fast != NULL && fast->link != NULL) {
4         slow = slow->link;          // 1 step
▶         fast = fast->link->link;    // 2 steps
6         if (slow == fast) return true;
7     }
8     return false;
9 }
```

VARIABLES

`slow = &node(2)`

`fast = &node(3)`

Code Trace: `has_cycle()` — Tortoise and Hare



5 → 3: cycle

Not on the same node; continue.

Step 5 / 13

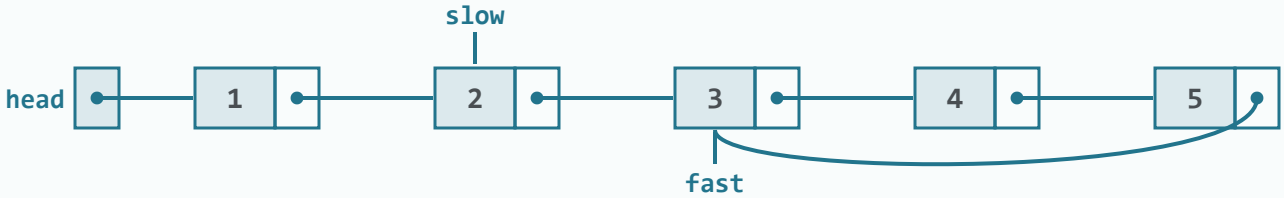
```
1 bool has_cycle(node_ptr head) {
2     node_ptr slow = head, fast = head;
3     while (fast != NULL && fast->link != NULL) {
4         slow = slow->link;          // 1 step
5         fast = fast->link->link;    // 2 steps
▶         if (slow == fast) return true;
7     }
8     return false;
9 }
```

VARIABLES

`slow` = **&node(2)**

`fast` = **&node(3)**

Code Trace: `has_cycle()` — Tortoise and Hare



5 → 3: cycle

`fast` and its successor are not NULL: keep running.

Step 6 / 13

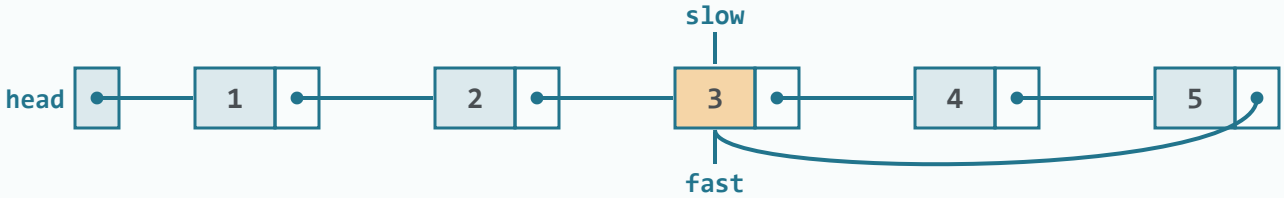
```
1 bool has_cycle(node_ptr head) {
2     node_ptr slow = head, fast = head;
▶     while (fast != NULL && fast->link != NULL) {
4         slow = slow->link;          // 1 step
5         fast = fast->link->link;    // 2 steps
6         if (slow == fast) return true;
7     }
8     return false;
9 }
```

VARIABLES

`slow` = **&node(2)**

`fast` = **&node(3)**

Code Trace: `has_cycle()` — Tortoise and Hare



5 → 3: cycle

Round 2: the tortoise takes 1 step → node(3).

Step 7 / 13

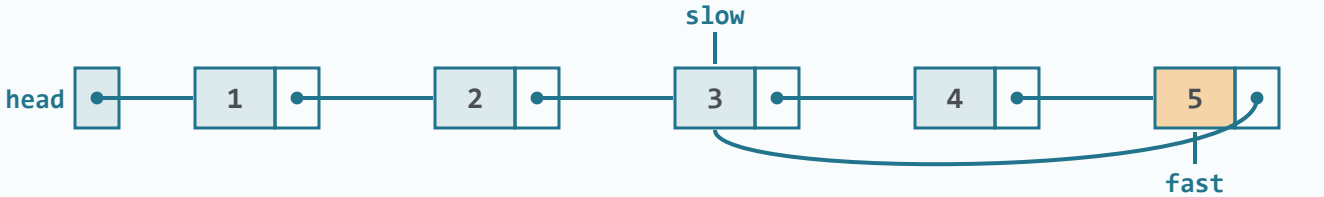
```
1 bool has_cycle(node_ptr head) {
2     node_ptr slow = head, fast = head;
3     while (fast != NULL && fast->link != NULL) {
▶         slow = slow->link;          // 1 step
5         fast = fast->link->link;    // 2 steps
6         if (slow == fast) return true;
7     }
8     return false;
9 }
```

VARIABLES

`slow = &node(3)`

`fast = &node(3)`

Code Trace: `has_cycle()` — Tortoise and Hare



5 → 3: cycle

The hare takes 2 steps → node(5).

Step 8 / 13

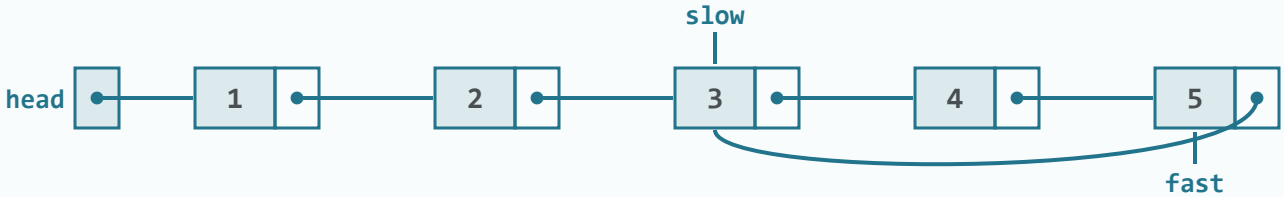
```
1 bool has_cycle(node_ptr head) {
2     node_ptr slow = head, fast = head;
3     while (fast != NULL && fast->link != NULL) {
4         slow = slow->link;          // 1 step
▶         fast = fast->link->link;    // 2 steps
6         if (slow == fast) return true;
7     }
8     return false;
9 }
```

VARIABLES

`slow = &node(3)`

`fast = &node(5)`

Code Trace: `has_cycle()` — Tortoise and Hare



5 → 3: cycle

Not on the same node; continue.

Step 9 / 13

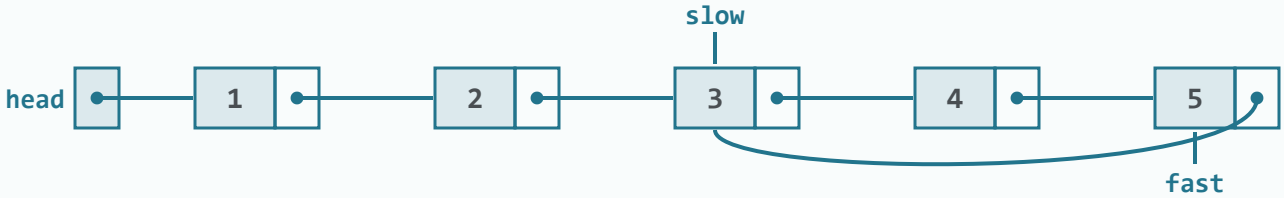
```
1 bool has_cycle(node_ptr head) {
2     node_ptr slow = head, fast = head;
3     while (fast != NULL && fast->link != NULL) {
4         slow = slow->link;          // 1 step
5         fast = fast->link->link;    // 2 steps
▶         if (slow == fast) return true;
7     }
8     return false;
9 }
```

VARIABLES

`slow` = **&node(3)**

`fast` = **&node(5)**

Code Trace: `has_cycle()` — Tortoise and Hare



5 → 3: cycle

`fast` and its successor are not NULL: keep running.

Step 10 / 13

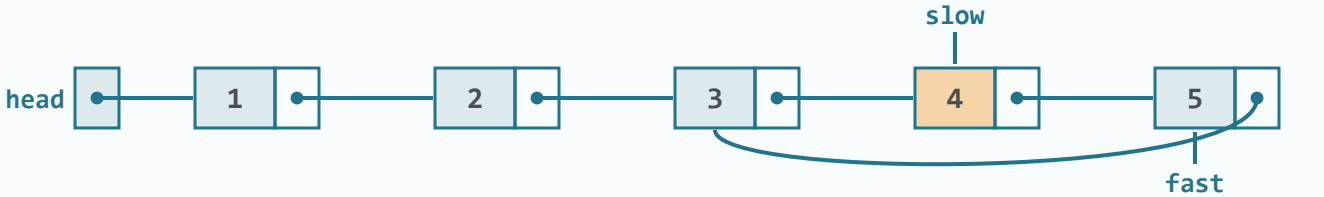
```
1 bool has_cycle(node_ptr head) {
2     node_ptr slow = head, fast = head;
▶     while (fast != NULL && fast->link != NULL) {
4         slow = slow->link;          // 1 step
5         fast = fast->link->link;    // 2 steps
6         if (slow == fast) return true;
7     }
8     return false;
9 }
```

VARIABLES

`slow = &node(3)`

`fast = &node(5)`

Code Trace: `has_cycle()` — Tortoise and Hare



5 → 3: cycle

Round 3: the tortoise takes 1 step → node(4).

Step 11 / 13

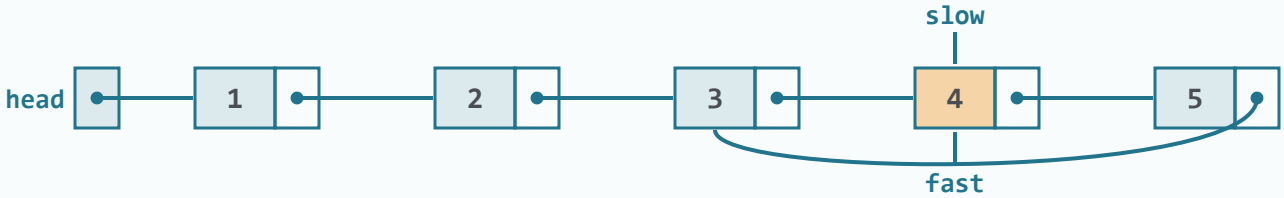
```
1 bool has_cycle(node_ptr head) {
2     node_ptr slow = head, fast = head;
3     while (fast != NULL && fast->link != NULL) {
          slow = slow->link;          // 1 step
5         fast = fast->link->link;    // 2 steps
6         if (slow == fast) return true;
7     }
8     return false;
9 }
```

VARIABLES

`slow = &node(4)`

`fast = &node(5)`

Code Trace: `has_cycle()` — Tortoise and Hare



5 → 3: cycle

The hare takes 2 steps → node(4).

Step 12 / 13

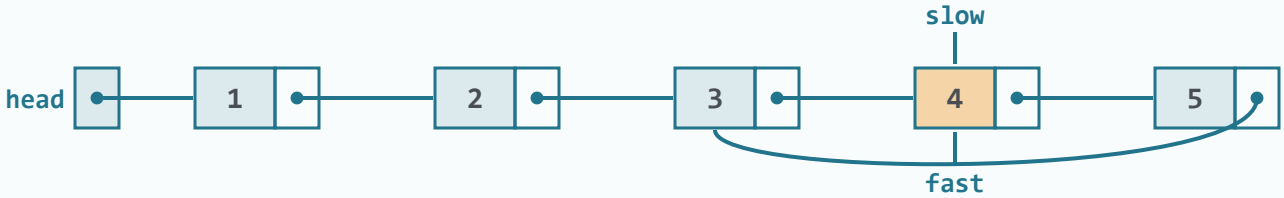
```
1 bool has_cycle(node_ptr head) {
2     node_ptr slow = head, fast = head;
3     while (fast != NULL && fast->link != NULL) {
4         slow = slow->link;          // 1 step
▶         fast = fast->link->link;    // 2 steps
6         if (slow == fast) return true;
7     }
8     return false;
9 }
```

VARIABLES

`slow = &node(4)`

`fast = &node(4)`

Code Trace: `has_cycle()` — Tortoise and Hare



5 → 3: cycle

They meet! The hare lapped the cycle and caught the tortoise → there is a cycle, `true`.

Step 13 / 13

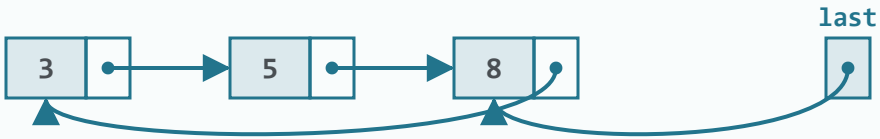
```
1 bool has_cycle(node_ptr head) {
2     node_ptr slow = head, fast = head;
3     while (fast != NULL && fast->link != NULL) {
4         slow = slow->link;          // 1 step
5         fast = fast->link->link;    // 2 steps
▶         if (slow == fast) return true;
7     }
8     return false;
9 }
```

VARIABLES

`slow = &node(4)`

`fast = &node(4)`

A Circular List: The Last Node Points to the First




```c
int c_length(node_ptr last) {
    int n = 0;
    node_ptr p = last;
    if (last == NULL) return 0;  // empty list
    do {
        n++;
        p = p->link;
    } while (p != last);         // full circle
    return n;
}
```

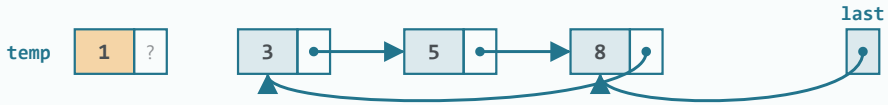
Why `last`?

`last→link` is the first node: inserting at both the front and the back is O(1).

No NULL

A traversal ends by coming back around, not at `p == NULL`; hence `do-while`.

Inserting at the Front and Back of a Circular List



3 → 5 → 8 → (3), `last` = 8. We insert the new node 1.

Step 1 / 4

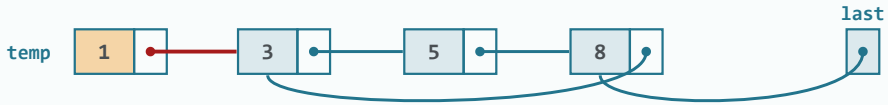
```
▶ void c_insert_front(node_ptr *last, node_ptr temp) {
2     if (*last == NULL) {          // empty list
3         *last = temp;
4         temp->link = temp;        // points to itself
5     } else {
6         temp->link = (*last)->link;  // old first node
7         (*last)->link = temp;
8     }
9 }
```

VARIABLES

`*last = &node(8)` `temp = &node(1)`

Back insertion: `c_insert_front(last, temp); *last = temp;` — front and back insertion are the same link operation; only the last pointer differs.

Inserting at the Front and Back of a Circular List



`temp→link = last→link` : the new node points to the old first node (3).

Step 2 / 4

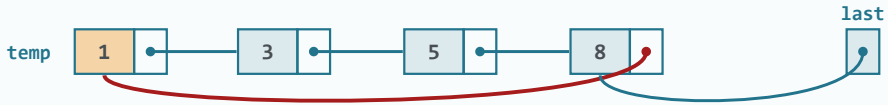
```
1 void c_insert_front(node_ptr *last, node_ptr temp) {
2     if (*last == NULL) {          // empty list
3         *last = temp;
4         temp->link = temp;       // points to itself
5     } else {
▶         temp->link = (*last)->link;  // old first node
7         (*last)->link = temp;
8     }
9 }
```

VARIABLES

`*last = &node(8)` `temp = &node(1)`

Back insertion: `c_insert_front(last, temp); *last = temp;` — front and back insertion are the same link operation; only the last pointer differs.

Inserting at the Front and Back of a Circular List



`last→link = temp` : 8 now points to 1. **Front insertion done**: 1 → 3 → 5 → 8.

Step 3 / 4

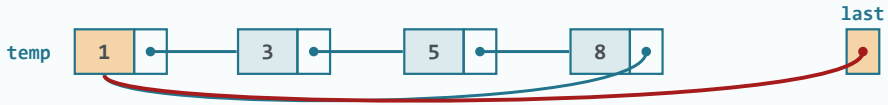
```
1 void c_insert_front(node_ptr *last, node_ptr temp) {
2     if (*last == NULL) {          // empty list
3         *last = temp;
4         temp->link = temp;       // points to itself
5     } else {
6         temp->link = (*last)->link;  // old first node
▶         (*last)->link = temp;
8     }
9 }
```

VARIABLES

`*last = &node(8)` `temp = &node(1)`

Back insertion: `c_insert_front(last, temp); *last = temp;` — front and back insertion are the same link operation; only the last pointer differs.

Inserting at the Front and Back of a Circular List



Back insertion = front insertion + `*last = temp` : last is now 1; order 3 → 5 → 8 → 1. Both are **O(1)**.

Step 4 / 4

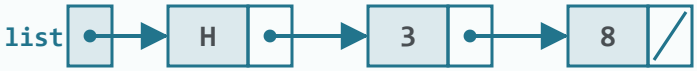
```
1 void c_insert_front(node_ptr *last, node_ptr temp) {
2     if (*last == NULL) {           // empty list
3         *last = temp;
4         temp->link = temp;        // points to itself
5     } else {
6         temp->link = (*last)->link;  // old first node
7         (*last)->link = temp;
8     }
▶ }
```

VARIABLES

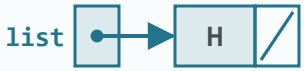
`*last = &node(1)` `temp = &node(1)`

Back insertion: `c_insert_front(last, temp); *last = temp;` — front and back insertion are the same link operation; only the last pointer differs.

The Header (Sentinel) Node



nonempty list: H → 3 → 8



empty list: only H

```
// sorted insert in a header list: no special case
void h_insert_sorted(node_ptr header, node_ptr temp) {
    node_ptr prev = header;          // never NULL
    while (prev->link != NULL &&
           prev->link->data < temp->data)
        prev = prev->link;
    temp->link = prev->link;
    prev->link = temp;
}
```

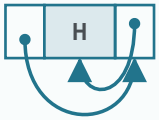
The idea

A node at the front that carries no data and is never deleted. Empty list = just the header.

The gain

Every real node has a predecessor: the `*head` and `if (prev == NULL)` special cases disappear.

The Doubly Linked List



empty: H points to itself



H ⇄ 4 ⇄ 8 ⇄ H

```
typedef struct dnode *dnode_ptr;
struct dnode {
    dnode_ptr llink; // previous
    int data;
    dnode_ptr rlink; // next
};
// invariant: p == p->llink->rlink == p->rlink->llink
```

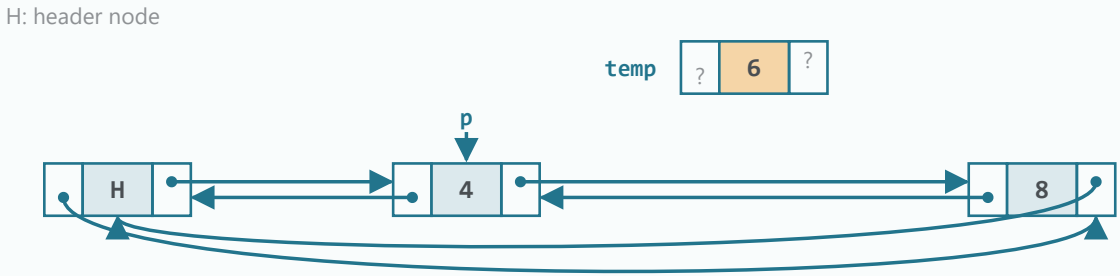
Pros

+ Traversal in both directions
+ O(1) deletion of a known node
+ O(1) insertion before a node

Cons

– 2 pointers per node
– 4 links per operation
– Easier to get a link wrong

Code Trace: `dinsert(p, temp)`



A circular doubly linked list with a header node: H ⇄ 4 ⇄ 8 ⇄ H. We insert 6 to the right of 4.

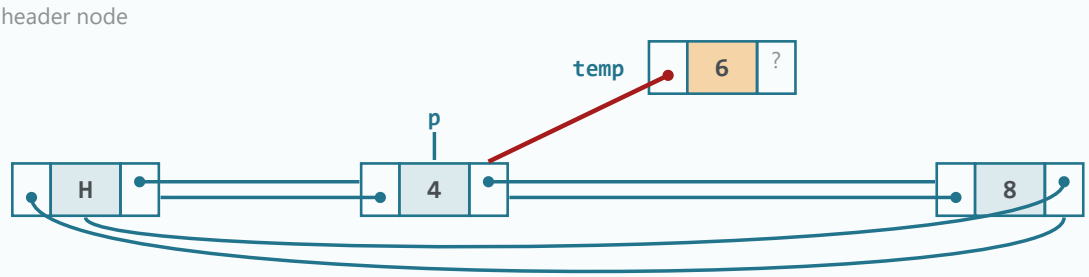
Step 1 / 6

```
1 // inserts temp to the right of node p
▶ void dinsert(dnode_ptr p, dnode_ptr temp) {
3     temp->llink = p;
4     temp->rlink = p->rlink;
5     p->rlink->llink = temp;
6     p->rlink = temp;
7 }
```

VARIABLES

- `p = &node(4)`
- `temp = &node(6)`
- `p→rlink = &node(8)`
- `temp→rlink = ?`

Code Trace: `dinsert(p, temp)`



`temp→llink = p` : the new node's left link points to 4.

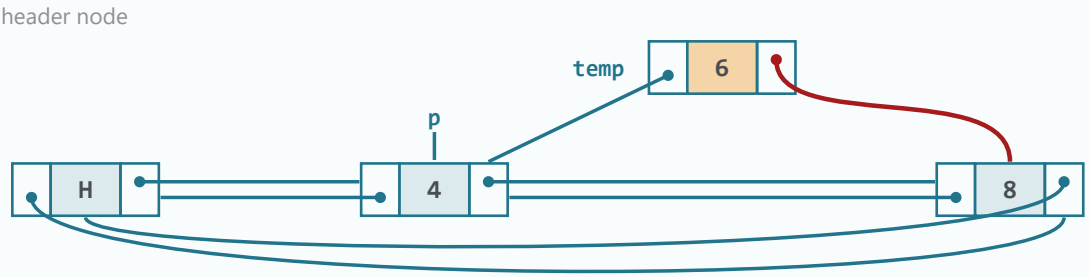
Step 2 / 6

```
1 // inserts temp to the right of node p
2 void dinsert(dnode_ptr p, dnode_ptr temp) {
▶     temp->llink = p;
4     temp->rlink = p->rlink;
5     p->rlink->llink = temp;
6     p->rlink = temp;
7 }
```

VARIABLES

- `p = &node(4)`
- `temp = &node(6)`
- `p→rlink = &node(8)`
- `temp→rlink = ?`

Code Trace: `dinsert(p, temp)`



`temp→rlink = p→rlink` : its right link points to 8. The list has not changed yet; only the new node is prepared.

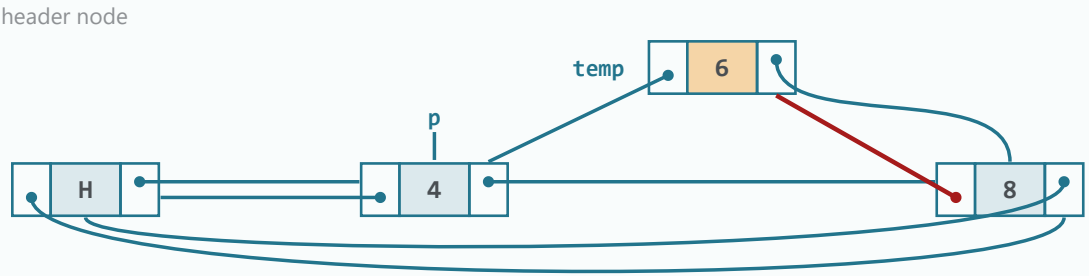
Step 3 / 6

```
1 // inserts temp to the right of node p
2 void dinsert(dnode_ptr p, dnode_ptr temp) {
3     temp->llink = p;
▶     temp->rlink = p->rlink;
5     p->rlink->llink = temp;
6     p->rlink = temp;
7 }
```

VARIABLES

`p = &node(4)` `temp = &node(6)` `p→rlink = &node(8)` `temp→rlink = &node(8)`

Code Trace: `dinsert(p, temp)`



`p→rlink→llink = temp` : 8's left link is now 6. (Since it relies on p->rlink still being 8, this line could not come **after** the 4th assignment, `p→rlink = temp` .)

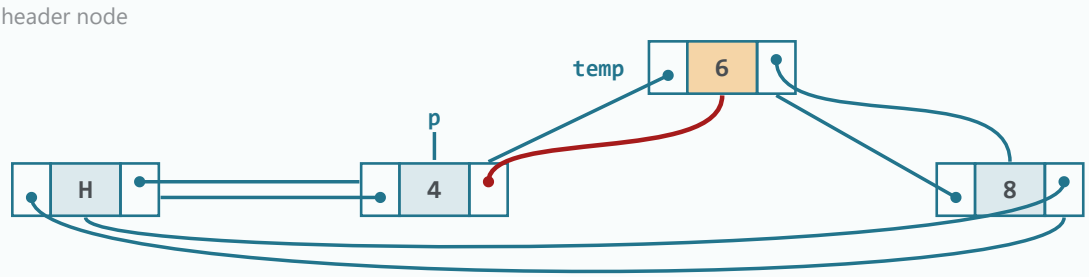
Step 4 / 6

```
1 // inserts temp to the right of node p
2 void dinsert(dnode_ptr p, dnode_ptr temp) {
3     temp->llink = p;
4     temp->rlink = p->rlink;
▶     p->rlink->llink = temp;
6     p->rlink = temp;
7 }
```

VARIABLES

- `p = &node(4)`
- `temp = &node(6)`
- `p→rlink = &node(8)`
- `temp→rlink = &node(8)`

Code Trace: `dinsert(p, temp)`



`p→rlink = temp` : the last link. Four links updated, no special case: **O(1)**.

Step 5 / 6

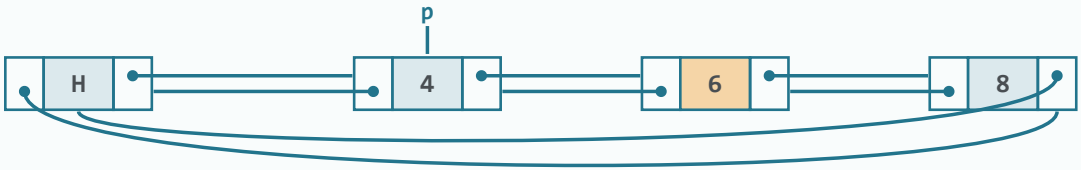
```
1 // inserts temp to the right of node p
2 void dinsert(dnode_ptr p, dnode_ptr temp) {
3     temp->llink = p;
4     temp->rlink = p->rlink;
5     p->rlink->llink = temp;
▶     p->rlink = temp;
7 }
```

VARIABLES

- `p = &node(4)`
- `temp = &node(6)`
- `p→rlink = &node(6)`
- `temp→rlink = &node(8)`

Code Trace: `dinsert(p, temp)`

H: header node



Result: H ⇄ 4 ⇄ 6 ⇄ 8 ⇄ H. Thanks to the header node, the same code works on an empty list and at the end.

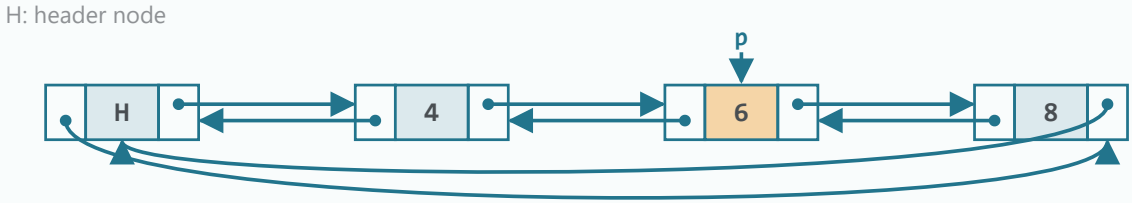
Step 6 / 6

```
1 // inserts temp to the right of node p
2 void dinsert(dnode_ptr p, dnode_ptr temp) {
3     temp->llink = p;
4     temp->rlink = p->rlink;
5     p->rlink->llink = temp;
▶     p->rlink = temp;
7 }
```

VARIABLES

- `p = &node(4)`
- `temp = &node(6)`
- `p→rlink = &node(6)`
- `temp→rlink = &node(8)`

Deleting from a Doubly Linked List: No Search for the Predecessor



H ⇄ 4 ⇄ 6 ⇄ 8 ⇄ H; `p` = node(6). Its predecessor is already at hand as `p→llink`.

Step 1 / 4

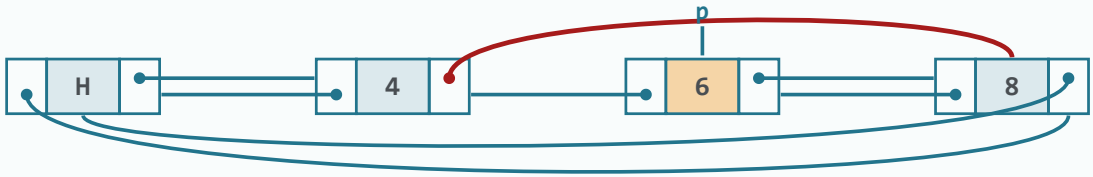
```
1 void ddelete(dnode_ptr header, dnode_ptr p) {
▶     if (p == header) {        // never delete header
3         fprintf(stderr, "Cannot delete header\n");
4         return;
5     }
6     p->llink->rlink = p->rlink;
7     p->rlink->llink = p->llink;
8     free(p);
9 }
```

VARIABLES

p = &node(6)

Deleting from a Doubly Linked List: No Search for the Predecessor

H: header node



`p->llink->rlink = p->rlink` : 4's right is now 8.

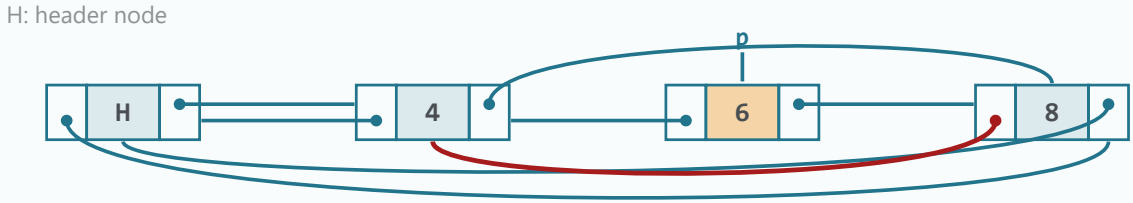
Step 2 / 4

```
1 void ddelete(dnode_ptr header, dnode_ptr p) {
2     if (p == header) {        // never delete header
3         fprintf(stderr, "Cannot delete header\n");
4         return;
5     }
▶     p->llink->rlink = p->rlink;
7     p->rlink->llink = p->llink;
8     free(p);
9 }
```

VARIABLES

p = &node(6)

Deleting from a Doubly Linked List: No Search for the Predecessor



`p->rlink->llink = p->llink` : 8's left is now 4.

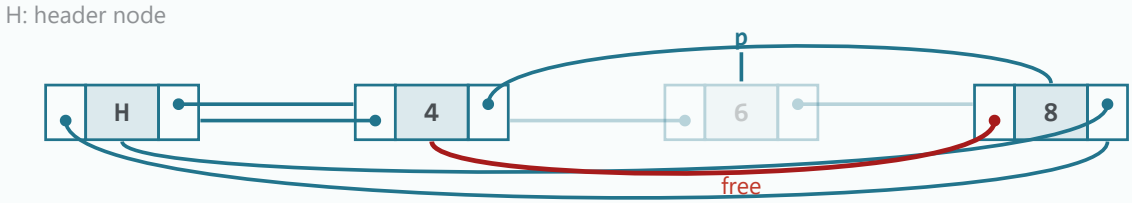
Step 3 / 4

```
1 void ddelete(dnode_ptr header, dnode_ptr p) {
2     if (p == header) {        // never delete header
3         fprintf(stderr, "Cannot delete header\n");
4         return;
5     }
6     p->llink->rlink = p->rlink;
▶     p->rlink->llink = p->llink;
8     free(p);
9 }
```

VARIABLES

p = &node(6)

Deleting from a Doubly Linked List: No Search for the Predecessor



`free(p)`. No search: **O(1)**. Thanks to the header, the first/last node needs no separate code.

Step 4 / 4

```
1 void ddelete(dnode_ptr header, dnode_ptr p) {
2     if (p == header) {       // never delete header
3         fprintf(stderr, "Cannot delete header\n");
4         return;
5     }
6     p->llink->rlink = p->rlink;
7     p->rlink->llink = p->llink;
▶     free(p);
9 }
```

VARIABLES

p = (freed)

The Available Space List

```
node_ptr avail = NULL;        // available nodes

node_ptr get_node(void) {
    node_ptr p;
    if (avail != NULL) {         // take from the list
        p = avail;
        avail = avail->link;
    } else {                     // else ask the system
        p = malloc(sizeof *p);
        if (p == NULL) exit(1);
    }
    return p;
}

void ret_node(node_ptr p) {      // instead of free
    p->link = avail;
    avail = p;
}
```

The idea (Horowitz)

Deleted nodes go onto the `avail` chain instead of to `free`; new nodes are taken from there first.

Why?

Programs that insert and delete a lot make fewer `malloc` / `free` calls; getting/returning a node is O(1).

Limitation

Nodes on avail are not returned to the heap until the program ends; useful when there is a single node type.

`get_node` and `ret_node` Step by Step



The `avail` chain holds two free nodes; on the right, node(9) is in use.

Step 1 / 5

In fact `avail` is a **stack**: `get_node` = pop, `ret_node` = push.

`get_node` and `ret_node` Step by Step



`get_node()` : avail is not empty → `p = avail` . No malloc call.

In fact `avail` is a **stack**: `get_node` = pop, `ret_node` = push.

`get_node` and `ret_node` Step by Step

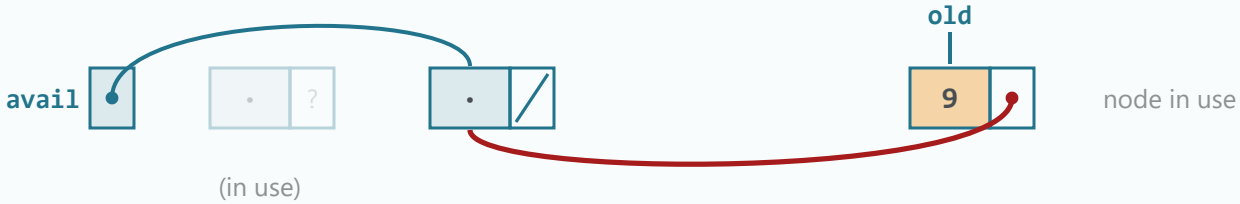


`avail = avail→link` : the head of the chain moves on; p now belongs to the caller.

Step 3 / 5

In fact `avail` is a **stack**: `get_node` = pop, `ret_node` = push.

`get_node` and `ret_node` Step by Step

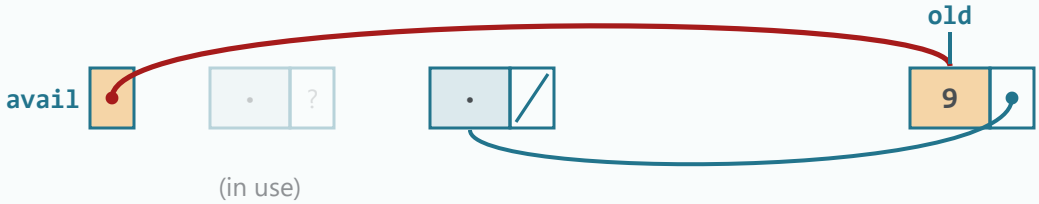


`ret_node(old)` : `old→link = avail` .

Step 4 / 5

In fact `avail` is a **stack**: `get_node` = pop, `ret_node` = push.

`get_node` and `ret_node` Step by Step



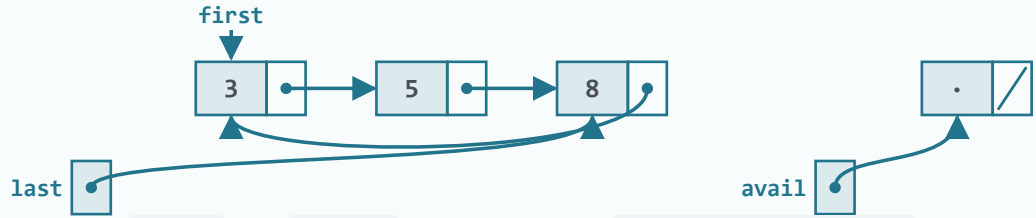
(in use)

`avail = old` : the node is ready to be reused. Both are **O(1)**, like push/pop on a linked stack.

Step 5 / 5

In fact `avail` is a **stack**: `get_node` = pop, `ret_node` = push.

Returning a Circular List in One Move (Horowitz: `cerase`)



3 → 5 → 8 → (3), `last` = 8; `avail` holds one node. `first = last→link` = 3.

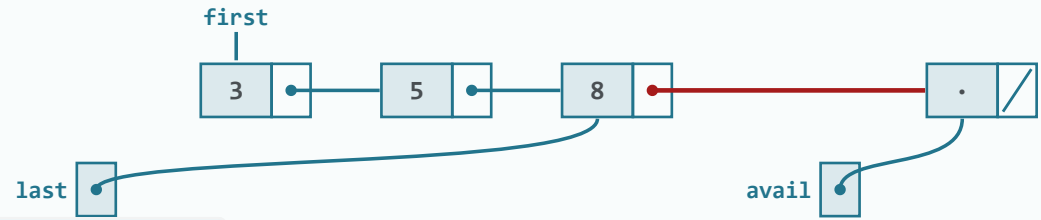
Step 1 / 4

```
1  // return a circular list to avail in O(1)
2  void cerase(node_ptr *last) {
3      node_ptr first;
4      if (*last != NULL) {
▶          first = (*last)->link;  // first node
6          (*last)->link = avail;  // open the ring
7          avail = first;
8          *last = NULL;
9      }
10 }
```

VARIABLES

`first` = **&node(3)** `*last` = **&node(8)** `avail` = **&node(·)**

Returning a Circular List in One Move (Horowitz: `cerase`)



`last→link = avail` : the ring is opened and its end joined to the avail chain.

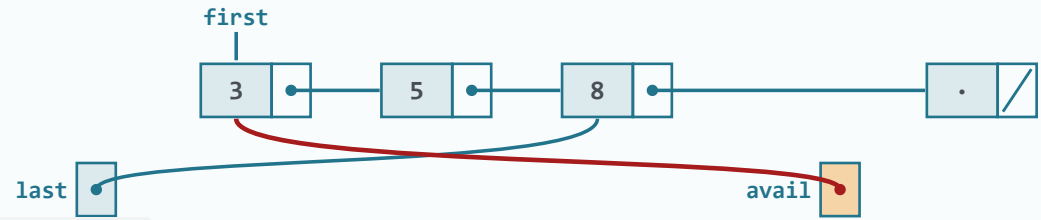
Step 2 / 4

```
1  // return a circular list to avail in O(1)
2  void cerase(node_ptr *last) {
3      node_ptr first;
4      if (*last != NULL) {
5          first = (*last)->link;  // first node
▶          (*last)->link = avail;  // open the ring
7          avail = first;
8          *last = NULL;
9      }
10 }
```

VARIABLES

`first = &node(3)` `*last = &node(8)` `avail = &node(·)`

Returning a Circular List in One Move (Horowitz: `cerase`)



`avail = first` : the whole list now sits in front of avail.

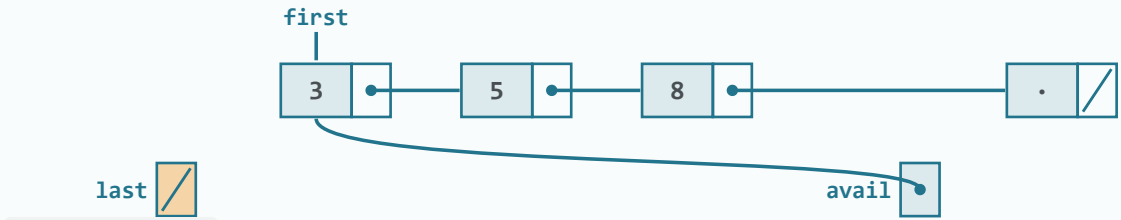
Step 3 / 4

```
1  // return a circular list to avail in O(1)
2  void cerase(node_ptr *last) {
3      node_ptr first;
4      if (*last != NULL) {
5          first = (*last)->link;  // first node
6          (*last)->link = avail;  // open the ring
▶          avail = first;
8          *last = NULL;
9      }
10 }
```

VARIABLES

`first = &node(3)` `*last = &node(8)` `avail = &node(3)`

Returning a Circular List in One Move (Horowitz: `cerase`)



`*last = NULL`. **O(1)** whatever the length; for a plain singly linked list, finding the end would take O(n).

Step 4 / 4

```
1  // return a circular list to avail in O(1)
2  void cerase(node_ptr *last) {
3      node_ptr first;
4      if (*last != NULL) {
5          first = (*last)->link;  // first node
6          (*last)->link = avail;  // open the ring
7          avail = first;
▶          *last = NULL;
9      }
10 }
```

VARIABLES

`first = &node(3)` `*last = NULL` `avail = &node(3)`

LAB · C CODE STEP BY STEP

Lab: Singly Linked List

Sorted insertion and deletion

Input (number = sorted insert, delete 30) `30, 10, 20, 25, delete 10, delete 5` Apply

CODE: **insert_sorted**

```
 1 void insert_sorted(node_ptr *head,
 2                     node_ptr temp) {
 3   node_ptr prev = NULL, p = *head;
 4   while (p != NULL &&
 5          p->data < temp->data) {
 6     prev = p;
 7     p = p->link;
 8   }
 9   temp->link = p;
10   if (prev == NULL) *head = temp;  // at fro…
11   else prev->link = temp;   // middle/end
12 }
```

empty list (NULL)

`head` NULL

OUTPUT (none yet)

The list starts empty: `head = NULL`. `insert_sorted` inserts the numbers keeping the list in increasing order; `delete_key` searches for a value and unlinks its node.

Complexity Summary

Operation	Array	Singly linked (head)	Circular (last)	Doubly linked (header)
Access the i-th element	O(1)	O(i)	O(i)	O(i)
Insert at front	O(n)	O(1)	O(1)	O(1)
Insert at back	O(1)*	O(n)	O(1)	O(1)
Insert after a known node	O(n)	O(1)	O(1)	O(1)
Delete a known node	O(n)	O(n)†	O(n)†	O(1)
Search by value	O(n)	O(n)	O(n)	O(n)
Concatenate	O(n+m)	O(n)	O(1)	O(1)
Extra space per node	—	1 pointer	1 pointer	2 pointers

* needs resizing when full · † a traversal to find the predecessor; O(1) if it is known. Floyd, middle node, reversal: O(n) time, O(1) extra space.

WRAP-UP

Common Mistakes

1. Overwriting a link before saving it: write `p→link = temp` first and the rest of the list is lost.

2. Passing the pointer by value to a routine that changes the head (`node_ptr head`); the list in main is not updated.

3. Not checking the result of `malloc`; forgetting `free` (a leak) or calling it twice.

4. Reading the `link` of a freed node; save the successor first, then free.

5. Not testing the empty list, a single node, and the head/tail cases.

6. Traversing a circular list with `p ≠ NULL`: the loop never ends.

Exercises

1. Merge two sorted lists into one sorted list without allocating nodes.
2. Find the k-th node from the end of a list in a single pass.
3. Delete the duplicates from a sorted list; return the deleted nodes with `ret_node`.
4. Write `invert` recursively. What is the extra space?
5. After Floyd finds a cycle, compute the length of the cycle.

▶ **Answers / hints**

Instructor "If I hold your homework over mine against the light… (there should be no difference)"

WRAP-UP

Chapter Summary

Node + link

List elements are scattered; the order is built with `link` fields. Empty list = NULL.

Link discipline

Save first, link next, release (`free`) last. Pass by address when the head changes.

Costs

Insert/delete O(1) when the position is known; access and search O(n).

Classics

Reversal, concatenation, middle node, Floyd: one pass, O(1) extra space.

Kinds

Circular (last pointer), doubly linked (two directions), header node (no special cases).

Memory

The avail list reuses nodes; a circular list is returned in O(1).

References and Next Steps

- E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2nd ed., Chapter 4 (lists, avail, circular and doubly linked lists).
- T. H. Cormen et al., *Introduction to Algorithms*, Section 10.2 (lists with sentinels).
- D. E. Knuth, *The Art of Computer Programming* Vol. 2, Exercise 3.1-6 (Floyd's cycle detection).

← Chapter 3: Recursion, Binary Search and Backtracking

Course page

Chapter 5: Polynomials and Sparse Matrices

The End

Chapter 4: Linked Lists

Next chapter: Chapter 5: Polynomials and Sparse Matrices · Course page