Lecture 1 — Foundations: C++ Refresher, Memory, and Arrays

Course: Data Structure (EF234201)

Book: Irfan Subakti, *Data Structure with C++, Java and Python*, Chapter 1

Lecturer: Irfan Subakti — yifana@gmail.com

Today we build the ground floor of the course: what a data structure is, how memory holds data, the C++ we will use all semester, and the first structure of all, the array. Book: Chapter 1, Sections 1.1–1.10.

What This Meeting Covers

1. **Why data structures** — Pustaka Ceria, interface and implementation, and why the choice beats a faster computer.
2. **Memory** — bits, bytes, addresses, types, little-endian order, the stack and the heap.
3. **A C++ refresher** — types, input and output, decisions and loops, functions, structs, and the operators `&` and `*`.
4. **Arrays in depth** — the address formula, records, bounds, traversals, row-major tables, `std::vector`, `ArrayList` and `list`.
5. **A first taste of complexity** — $O(1)$ access versus $O(n)$ insertion, big-O, amortized cost; and the workbench of VS Code with three languages.

Every program today exists in C++, Java and Python, and every output shown is the output the program really printed. Book: Chapter 1.

Where We Begin: a Library Without Shelves

Pustaka Ceria, a small campus library, opens with eight books stacked on the floor. Any method finds a book among eight.

A year later it owns eighty thousand books. Reading every cover now costs forty thousand looks on average.

The library buys shelves, numbers them, sorts the books by code and keeps a catalog. The books did not change; their **organization** did.

That organization is a *data structure*. Pustaka Ceria stays with us all semester: an array today, then stacks, queues, trees, graphs and hash tables.

From the programming course before this one we bring variables, loops and functions; what is new is thinking about how many values a program keeps and how it keeps them. Book: Section 1.1.1.

What a Data Structure Is

> **DEFINITION.** A *data structure* is a way of storing and organizing a collection of data in a computer's memory, together with the operations that can be performed on it, so that those operations are carried out correctly and efficiently. The five usual operations are **insert**, **delete**, **search**, **traverse** and **sort**.

Interface — the operations and what each promises: "I can give you the book B1005, or tell you it is not here". A type described only this way is an *abstract data type* (ADT).

Implementation — the bytes in memory and the steps that keep the promise: a sorted row searched by halving, or eight bins chosen by the last digit.

The interface is what the visitor needs to know; the implementation is what the librarian needs to know. A good structure lets the librarian change the implementation without the visitor noticing. Book: Section 1.1.2.

Why the Choice Matters

Table 1.1 counts the looks needed to find one book among n with three different organizations.

Table 1.1 Looks needed in the worst case to find one book among n, for three ways of keeping the books.

Books n	Read every cover (n looks)	Halve each time ($\approx \log_2 n$ looks)	Compute the position (about 1 look)
8	8	3	1
1,000	1,000	10	1
80,000	80,000	17	1
1,000,000	1,000,000	20	1
1,000,000,000	1,000,000,000	30	1

At one microsecond a look, a billion books cost seventeen minutes by the first method and thirty microseconds by the second. A computer a hundred times faster still loses. Choosing the right data structure is usually worth more than buying the fastest computer. Book: Section 1.1.3.

Data Structures Behind Every Web Page

The **Internet** is a network of networks speaking TCP/IP; the **World Wide Web** is one service on it, where a browser asks a server for documents over HTTP.

The browser's name cache is a **hash table**; the name servers form a **tree**; routers keep waiting requests in **queues** and choose paths over a **graph**.

The server reads files listed in **arrays** and indexed by **trees**; the browser builds a **tree** of elements and walks it with a **stack**.

Five steps of one page request, at least six data structures, each chosen because it makes that step cheap, and every one of them is in the book. Book: Section 1.1.4.

Six Families of Data Structures

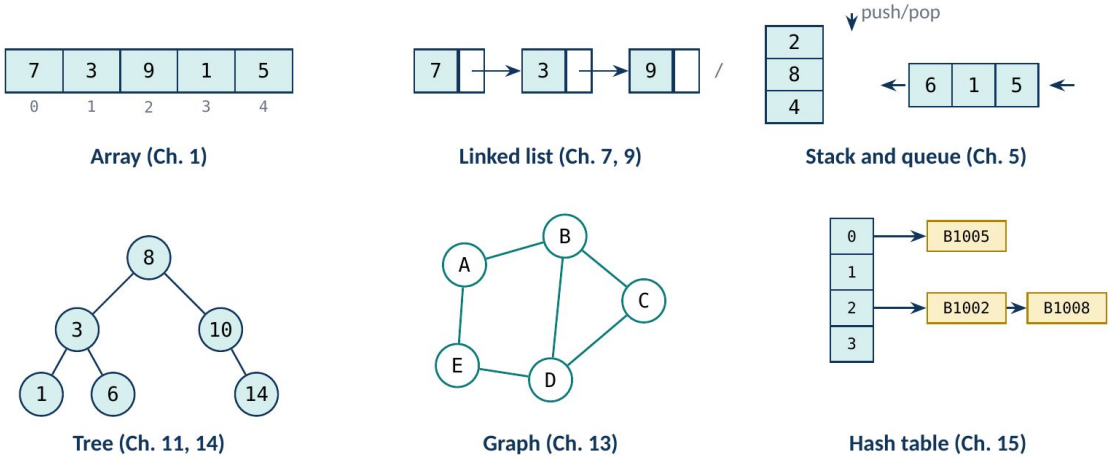


Figure 1.1 The six families of data structures in this book: arrays, linked lists, stacks and queues, trees (with heaps), graphs and hash tables, each labeled with the chapters that build it.

Figure 1.1: arrays side by side, linked lists of boxes that point onward, stacks and queues, trees and heaps, graphs of connections, and hash tables that compute where an item lives. Book: Section 1.1.5.

Three Languages, One Idea

Table 1.2 The ten most popular programming languages in the TIOBE Index of September 2026.

Rank	Language	Rating
1	Python	17.76%
2	C	10.28%
3	C++	8.67%
4	Java	7.54%
5	C#	4.22%
6	JavaScript	2.76%
7	Visual Basic	2.55%
8	SQL	2.16%
9	R	1.69%
10	Rust	1.34%

***C++** hides nothing: an array of eight books really sits side by side. **Java** puts every object on the heap; **Python** makes every variable a name for an object. Table 1.2 measures popularity, not quality. Book: Section 1.2.1.*

From Source File to Running Program

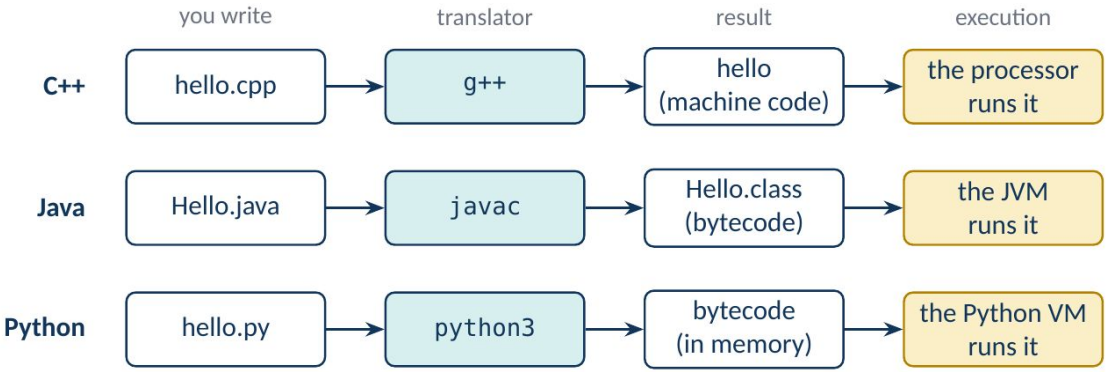


Figure 1.2 From source file to running program: C++ is compiled to machine code, Java to bytecode for the Java Virtual Machine, and Python to bytecode that its own virtual machine runs at once.

Figure 1.2: C++ is compiled to machine code, Java to bytecode for the Java Virtual Machine, and Python to bytecode that its own virtual machine runs at once. Book: Section 1.2.2.

Bits, Bytes and Hexadecimal

A **bit** is a switch, 0 or 1. Each extra bit doubles the number of states, so k bits can be in the number of states of Equation (1.1).

$$2^k \text{ different states} \tag{1.1}$$

A **byte** is eight bits: $2^8 = 256$ states, the numbers 0 to 255. The byte 11101010 is $128+64+32+8+2=234$.

Hexadecimal writes four bits as one digit, `0`–`9` then `a`–`f`: 1110 1010 is `ea`, written `0xEA` in all three languages.

*Memory is one long row of bytes, each with a numbered **address**; any address is reached in the same short time. That is random access, and it is what makes the array fast. Book: Sections 1.3.1 and 1.3.2.*

A Type Gives the Bytes a Meaning

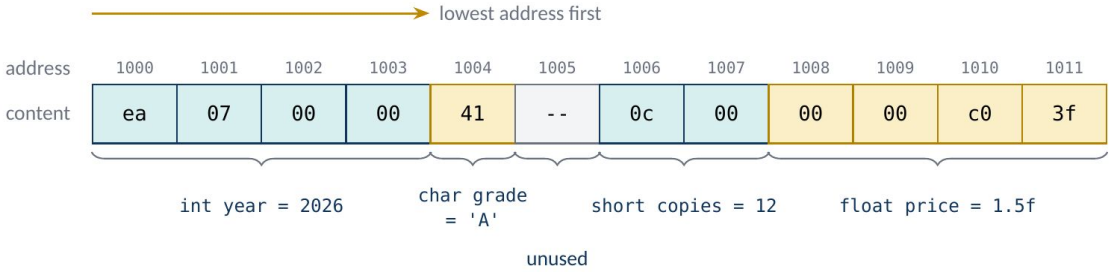


Figure 1.3 Twelve bytes of memory holding an `int`, a `char`, a `short` and a `float`; each type decides how many bytes a value uses and how they are read, and the lowest-addressed byte comes first.

*In Figure 1.3, 2026 is `0x000007EA`, stored least significant byte first: **little-endian**. Byte 1005 is **padding**, so that the `short` starts at an even address. The memory does not know what it holds; only the program does, through its types. Book: Section 1.3.3.*

Seeing the Bytes for Yourself

Listing 1.1 looks at the four bytes of the `int` 2026 one at a time, lowest address first.

Listing 1.1 From `memory_basics.cpp`: the bytes of an `int` and the machine's byte order.

```
    int year = 2026;                        // 2026 = 0x000007EA
    const unsigned char* b = reinterpret_cast<const unsigned char*>(&year);
    cout << "\nThe int " << year << " occupies these bytes, lowest address first:";
    for (size_t i = 0; i < sizeof year; ++i) {
        cout << ' ' << hex << setw(2) << setfill('0') << int(b[i]);
    }
    cout << dec << '\n';
    cout << "So this machine is " << (b[0] == 0xEA ? "little" : "big")
         << "-endian.\n";
```

```
The int 2026 occupies these bytes, lowest address first: ea 07 00 00
So this machine is little-endian.
```

`sizeof` gives the bytes of a type; `&year` is the address of `year`; the cast lets us read the bytes one at a time. The sizes are those of a 64-bit machine; the standard fixes only that `char` is one byte, so the book always asks `sizeof`. Book: Section 1.3.4.

The Stack and the Heap

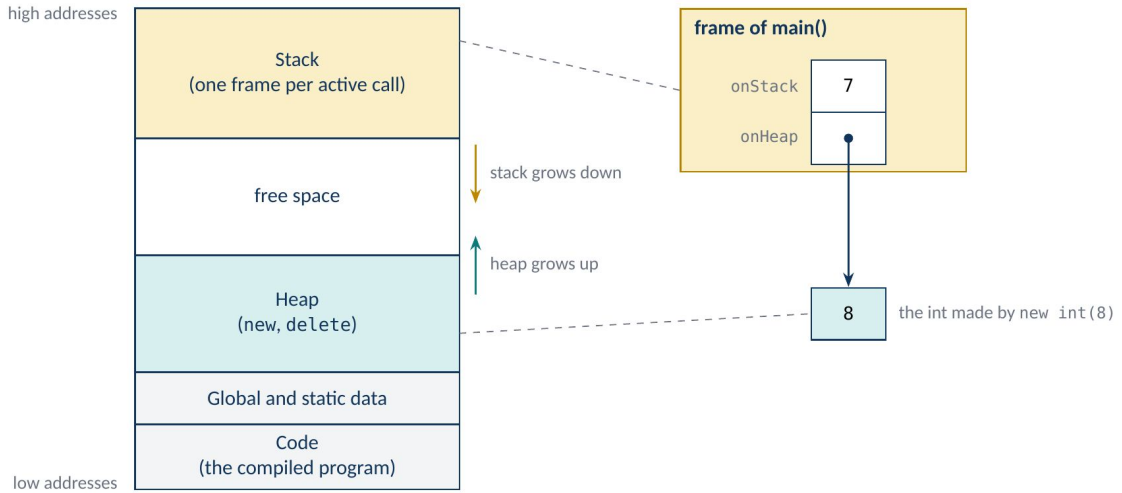


Figure 1.4 The memory of a running C++ program: code and global data at the bottom, the heap growing upward, the stack growing downward, and, on the right, the frame of `main` whose pointer `onHeap` refers to an `int` on the heap.

*Figure 1.4: each call gets a **stack frame**, released on return, last in first out; memory from `new` lives on the **heap** until `delete`, and every forgotten `delete` is a **memory leak**. `memory_basics.cpp` prints a stack address near `0x7ff...` and a heap address near `0x55...`. Java and Python put objects on the heap and free them with a **garbage collector**. Book: Section 1.3.5.*

The C++ Refresher: Types

Every program has the same skeleton: `#include`, `using namespace std;`, `int main()`, statements ending in `;`, and `return 0`.

Table 1.3 The basic C++ types used in this book, with their sizes on a 64-bit machine.

Type	Bytes (here)	Holds	Range or precision
`char`	1	one character code, or a small number	-128 to 127
`bool`	1	`true` or `false`	two values
`short`	2	a small whole number	$-32,768$ to $32,767$
`int`	4	a whole number	-2^{31} to $2^{31}-1$, about ± 2.1 billion
`long long`	8	a large whole number	-2^{63} to $2^{63}-1$, about $\pm 9.2 \times 10^{18}$
`float`	4	a real number	about 7 significant digits
`double`	8	a real number	about 15 to 16 significant digits
`size_t`	8	a size or an index, never negative	0 to $2^{64}-1$

Table 1.3 lists the sizes that `sizeof` printed on a 64-bit machine. Three habits: give every variable a value before reading it; choose a type large enough (overflow is silent); and remember that `7 / 2` is `3`, while `static_cast<double>(7) / 2` is `3.5`. Book: Sections 1.4.1 and 1.4.2.

Input and Output

Listing 1.2 From `io_sum.cpp`: reading a count and that many numbers with `cin`, and summarizing them.

```
    int n;
    if (!(cin >> n) || n <= 0) {
        cout << "Expected a positive count first.\n";
        return 1;
    }
    long long sum = 0;
    int smallest = 0, largest = 0;
    for (int i = 0; i < n; ++i) {
        int x;
        cin >> x;
        sum += x;
        if (i == 0 || x < smallest) smallest = x;
        if (i == 0 || x > largest) largest = x;
    }
    double average = static_cast<double>(sum) / n;
```

Listing 1.2 reads a count, then that many numbers; `<<` and `>>` point the way the data flows, and `if (!(cin >> n) || n <= 0)` reads and checks. Run it as `./io_sum < week_loans.txt` with the week `12 9 15 4 20 11 6`: total 77, average 11.00, smallest 4, largest 20. Java uses `Scanner`, Python `sys.stdin.read().split()`. Book: Section 1.4.3.

Decisions, Loops and Functions

`if`/`else`, `switch` (end each `case` with `break` or `return`), `while`, `do … while`, and `for`; conditions with `==`, `!=`, `&&`, `||`, `!`.

`for (int i = 0; i < n; i++)` runs exactly n times with `i` $= 0, \ldots, n-1$: precisely the valid indexes of an array of n elements.

A **function** has a signature, `long lateFee(int daysKept, char memberType)`; C++ passes parameters **by value**, so the function receives copies.

A **struct** groups related values under one name; a dot selects a field: `first.title`. An array of structs is the simplest table of records.

`break` leaves a loop at once; `continue` skips to the next round. The range-based `for (const Loan& loan : loans)` visits every element without an index. Book: Sections 1.4.4–1.4.6.

Worked Example 1: Late Fees — the Rule

Listing 1.3 From `late_fee.cpp`: the fee function with `if` and `switch`.

```
// Returns the fee in rupiah, or -1 for an unknown member type.
long lateFee(int daysKept, char memberType) {
    int daysLate = daysKept - LOAN_DAYS;
    if (daysLate <= 0) {
        return 0;
    }
    long base;
    if (daysLate <= 7) {
        base = daysLate * 1000L;          // Rp1,000 a day for the first week
    } else {
        base = 7 * 1000L + (daysLate - 7) * 2000L;   // then Rp2,000 a day
    }
    switch (memberType) {
        case 'S': return base / 2;        // students pay half
        case 'L': return base;
        case 'G': return base * 2;        // guests pay double
        default:  return -1;
    }
}
```

A loan lasts seven days; late costs Rp1,000 a day for the first week and Rp2,000 a day after; students pay half, guests double. In Listing 1.3, `-1` means "unknown member type". The `L` suffix on `1000L` makes the arithmetic happen in `long`. Book: Section 1.4.6.

Worked Example 1: the Loop Over Loans

`Loan loans[] = { {"Laskar Pelangi", 5, 'S'}, …, {"Sejarah Nusantara", 9, 'X'} };` holds six loans; Listing 1.4 totals their fees and skips the bad record.

Listing 1.4 The `main` of `late_fee.cpp`: an array of structs and a range-based `for` with `continue`.

```
    long total = 0;
    for (const Loan& loan : loans) {
        long fee = lateFee(loan.daysKept, loan.memberType);
        if (fee < 0) {
            cout << loan.title << ": unknown member type '" << loan.memberType
                 << "'\n";
            continue;                    // skip it, go on with the next loan
        }
        cout << loan.title << " (" << loan.memberType << ", " << loan.daysKept
             << " days): Rp" << fee << '\n';
        total += fee;
    }
    cout << "Total fees: Rp" << total << '\n';
```

The `struct Loan` above it has three fields: `title`, `daysKept` and `memberType`. Book: Section 1.4.6.

Worked Example 1: Check Two Lines by Hand

```
Laskar Pelangi (S, 5 days): Rp0
Bumi Manusia (S, 10 days): Rp1500
Filosofi Kopi (L, 10 days): Rp3000
Fisika Dasar (G, 16 days): Rp22000
Negeri 5 Menara (S, 20 days): Rp9500
Sejarah Nusantara: unknown member type 'X'
Total fees: Rp36000
```

Fisika Dasar, guest, 16 days: $16-7=9$ days late; base $7\times 1000+(9-7)\times 2000=11,000$; double: $22,000$.

Negeri 5 Menara, student, 20 days: 13 days late; base $7000+6\times 2000=19,000$; half: $9,500$.

Sejarah Nusantara, type `'X'`: `lateFee` returns -1; the loop reports it and continues, so one bad record does not spoil the total.

Working a program by hand is the most useful habit this course can teach. Book: Section 1.4.6.

The Same Rule in Python

Listing 1.5 is the Python version: a `dataclass` for the loan and a `match` statement for the member type.

Listing 1.5 From `late_fee.py`: the fee function with `match`.

```
def late_fee(days_kept: int, member_type: str) -> int:
    """Return the fee in rupiah, or -1 for an unknown member type."""
    days_late = days_kept - LOAN_DAYS
    if days_late <= 0:
        return 0
    if days_late <= 7:
        base = days_late * 1000          # Rp1,000 a day for the first week
    else:
        base = 7 * 1000 + (days_late - 7) * 2000   # then Rp2,000 a day
    match member_type:
        case "S":
            return base // 2             # students pay half
        case "L":
            return base
        case "G":
            return base * 2              # guests pay double
        case _:
            return -1
```

Python's `/` always gives a real number, so half the fee is `base // 2`. Java uses a `record` and a `switch`. All three print the same seven lines. `match` needs Python 3.10 or newer. Book: Section 1.4.6.

The Address-of Operator and a First Pointer

`&age` is the **address** of `age`; `int* p = &age;` declares a pointer; `*p` follows it (**dereferencing**). Listing 1.6 passes `age` both ways.

Listing 1.6 From `address_of.cpp`: a copy cannot change the caller's variable; an address can.

```
void birthdayByValue(int a) {              // a is a copy
    a = a + 1;
}

void birthdayByAddress(int* p) {           // p holds the caller's address
    *p = *p + 1;
}
    birthdayByValue(age);
    cout << "after birthdayByValue(age), age = " << age << '\n';
    birthdayByAddress(&age);
    cout << "after birthdayByAddress(&age), age = " << age << '\n';
```

```
after *p = 21, age = 21
after birthdayByValue(age), age = 21
after birthdayByAddress(&age), age = 22
```

*`sizeof(p)` is 8: every address takes eight bytes on a 64-bit machine. Java and Python never show addresses, but a variable of an object type holds a **reference**, and a method that receives one can change the object. Chapter 6 completes the story. Book: Section 1.4.7.*

The Array: One Row of Numbered Slots

> **DEFINITION.** An *array* is a fixed number of elements of the same type, stored one after another in a single contiguous block of memory, each identified by an integer *index*. The first index is 0, so an array of n elements has the indexes $0, 1, \ldots, n-1$.

`int loans[8] = {12, 9, 15, 4, 20, 11, 6, 15};` — the size is known when the program is compiled and never changes.

`int zeros[5] = {0};` sets every element to zero; `int loans[] = {…};` lets the compiler count.

`loans[3] = 5;` *puts 5 into the fourth place: index 3 is the fourth slot. Book: Section 1.5.1.*

The Address Formula

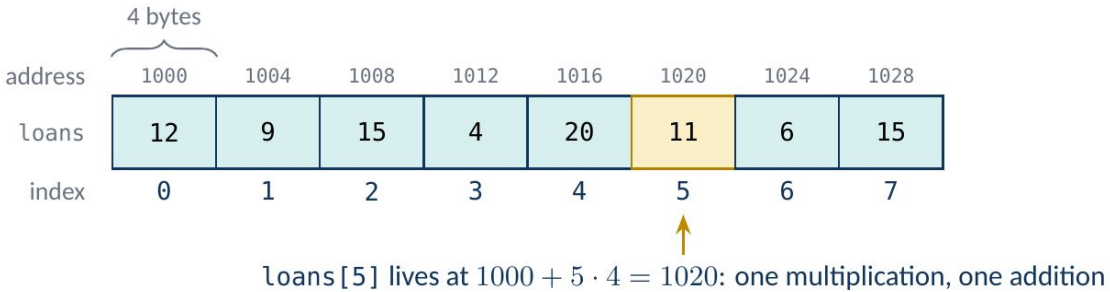


Figure 1.5 The array `loans` of eight `int`s in memory: each element takes 4 bytes, so element i begins at $1000+4\,i$.

$$\text{address}\big(A[i]\big) = b + i \cdot s \tag{1.2}$$

Equation (1.2) with $b=1000$ and $s=4$ in Figure 1.5: `loans[5]` begins at $1000+5\cdot 4=1020$. One multiplication and one addition, whether the array holds eight elements or eight million. Book: Section 1.5.1.

Why Indexes Start at 0

The index is not "which element in the count" but **how many elements past the start**: the first element is at $b+0\cdot s=b$.

Starting at 1 would cost one extra subtraction, $b+(i-1)\cdot s$, on every access, forever.

In C++, `loans[i]` is exactly `*(loans + i)`, and `&loans[i]` is exactly `loans + i`: adding i to a pointer moves it by i *elements*.

> **REMEMBER.** An array of n elements has **no** element with index n. The last one is $n-1$.

The square brackets are only a friendlier spelling of "start at the base, move i elements, follow the address". Book: Section 1.5.2.

An Array of Records: the Catalog

Listing 1.7 stores every field inside the record, so one `Book` is a solid block of 6+100+60+30=196 bytes.

Listing 1.7 From `book.h`: the `Book` record with fixed-size character arrays, and the helper that never writes past a field.

```
struct Book {
    char id[6];                          // "B1001" plus the terminating '\0'
    char title[100];
    char author[60];
    char category[30];
};

// Copies src into dst, cutting it short if needed; dst always ends with '\0'.
inline void safeCopy(char* dst, std::size_t dstSize, const char* src) {
    std::strncpy(dst, src, dstSize - 1);
    dst[dstSize - 1] = '\0';
}
```

A `std::string` would keep its characters elsewhere, on the heap. The price of fixed arrays is a limit: at most 99 characters in a title, because one `char` holds the terminating `'\0'`. `safeCopy` cuts a longer text short. Book: Section 1.5.3.

Eight Books End to End

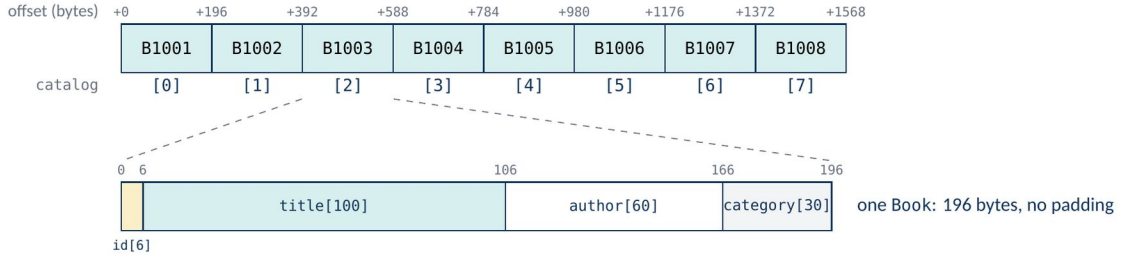


Figure 1.6 The catalog as eight 196-byte `Book` records laid end to end, and the inside of one record: its four fields begin at offsets 0, 6, 106 and 166.

Figure 1.6: the fields begin at offsets 0, 6, 106 and 166; consecutive books begin 196 bytes apart, as Equation (1.2) predicts with $s = 196$*. No padding, because a* `char` *may begin at any address. Book: Section 1.5.3.*

Measuring Where Each Book Begins

Listing 1.8 From `catalog_1d.cpp`: the byte offset of every element, and the check that `&catalog[i]` equals `catalog + i`.

```
    const char* base = reinterpret_cast<const char*>(&catalog[0]);
    std::printf("\n%-3s %-26s %s\n",
                "i", "&catalog[i] - base (bytes)", "== catalog + i ?");
    for (int i = 0; i < N; i++) {
        const char* here = reinterpret_cast<const char*>(&catalog[i]);
        long offset = static_cast<long>(here - base);
        bool same = (&catalog[i] == catalog + i);
        std::printf("%-3d %-26ld %s\n", i, offset, same ? "yes" : "NO");
    }
```

```
i   &catalog[i] - base (bytes) == catalog + i ?
0   0                         yes
1   196                       yes
2   392                       yes
3   588                       yes
4   784                       yes
5   980                       yes
6   1176                      yes
7   1372                      yes
```

Listing 1.8 measures each element's distance from `catalog[0]`. In Java, `new Book[8]` makes eight references, and each `new Book(...)` is a separate object on the heap; a Python list is the same. The formula still finds the reference in one step; one more step follows it to the book. Book: Section 1.5.3.

Staying Inside the Bounds

C++ does not check: `loans[8]` reads whatever lies at $b+8\cdot4$ — **undefined behavior**. The checked accessor `at(i)` throws `out_of_range` (Listing 1.9).

Listing 1.9 From `bounds.cpp`: indexing inside the bounds, and `at()` refusing an index that is out of range.

```
    for (size_t i : {3, 5}) {
        try {
            int value = v.at(i);          // throws if i is not a valid index
            cout << "v.at(" << i << ") = " << value << '\n';
        } catch (const out_of_range& e) {
            cout << "v.at(" << i << ") refused: " << e.what() << '\n';
        }
    }
```

```
v.at(3) = 40
v.at(5) refused: vector::_M_range_check: __n (which is 5) >= this->size() (which is 5)
```

Java throws `ArrayIndexOutOfBoundsException` and Python `IndexError`, at the exact line. Python also accepts negative indexes: `a[-1]` is the last element; in C++, `a[-1]` is the memory just before the array. Book: Section 1.5.4.

Walking an Array: Sum and Maximum

Listing 1.10 From `array_ops.cpp`: the sum, and the position of the largest element.

```
long sumOf(const int a[], int n) {
    long s = 0;
    for (int i = 0; i < n; i++) s += a[i];
    return s;
}

int indexOfMax(const int a[], int n) {
    int best = 0;
    for (int i = 1; i < n; i++) {
        if (a[i] > a[best]) best = i;
    }
    return best;
}
```

Listing 1.10: a traversal visits every element once, from index 0 to $n-1$. A parameter `const int a[]` is only a pointer, so the length travels as `n`; `sizeof loans / sizeof loans[0]` works only in `main`. `const` promises not to change the array. Book: Section 1.5.5.

Walking an Array: Search and Count

Listing 1.11 From `array_ops.cpp`: the first position of a value, and the number of times it occurs.

```
int findFirst(const int a[], int n, int key) {       // -1 means "not found"
    for (int i = 0; i < n; i++) {
        if (a[i] == key) return i;
    }
    return -1;
}

int countOf(const int a[], int n, int key) {
    int c = 0;
    for (int i = 0; i < n; i++) {
        if (a[i] == key) c++;
    }
    return c;
}
```

In Listing 1.11, `findFirst` returns -1 for "not found", which can never be a valid index: the contract of every search in Chapter 2. Book: Section 1.5.5.

Walking an Array: Reverse in Place

Listing 1.12 From `array_ops.cpp`: reversing the array with two indexes that meet in the middle.

```
void reverseInPlace(int a[], int n) {
    for (int i = 0, j = n - 1; i < j; i++, j--) {
        int t = a[i];
        a[i] = a[j];
        a[j] = t;
    }
}
```

```
loans      = [12, 9, 15, 4, 20, 11, 6, 15]
n          = 8
sum        = 92
max        = 20 at index 4
find 15    = 2
find 99    = -1
count 15   = 2
reversed   = [15, 6, 11, 20, 4, 15, 9, 12]
```

Listing 1.12: $\lfloor n/2 \rfloor$ *swaps, each through a temporary* `t`*. Python swaps in one line,* `a[i], a[j] = a[j], a[i]`*; Java and Python functions need no* `n`*, since arrays know* `a.length` *and lists* `len(a)`*. Book: Section 1.5.5.*

Two-Dimensional Arrays, Row-Major

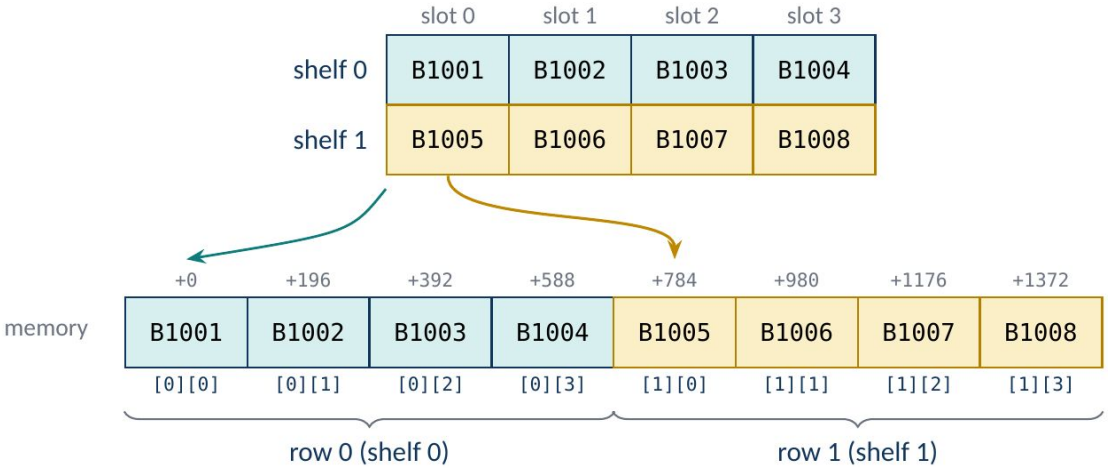


Figure 1.7 The 2 × 4 array `shelf` as a grid, and as it really lies in memory: row 0 first, then row 1, each element 196 bytes from the last.

$$\text{address}(A[r][c]) = b + (r \cdot C + c) \cdot s \tag{1.3}$$

Figure 1.7: C++ stores row 0 first, then row 1. Element $[r][c]$ is preceded by r whole rows of C elements and c elements of its own row, hence Equation (1.3). The number of rows does not appear. Book: Section 1.5.6.

Worked Example 2: Addresses in a Table

`int m[6][9]` begins at $b = 2000$ with $s = 4$; so $R = 6$ and $C = 9$.

Row-major, Equation (1.3): `m[4][7]` is at $2000 + (4 \cdot 9 + 7) \cdot 4 = 2000 + 43 \cdot 4 = 2172$.

Column-major, as Fortran stores it, Equation (1.4): $2000 + (7 \cdot 6 + 4) \cdot 4 = 2000 + 46 \cdot 4 = 2184$.

$$\text{address}(A[r][c]) = b + (c \cdot R + r) \cdot s \tag{1.4}$$

Which element begins at $2000 + 4 \cdot 50$? Position 50 is row $\lfloor 50/9 \rfloor = 5$, column $50 - 45 = 5$: `m[5][5]`.

Code that computes addresses by hand, or passes arrays between languages, must know which order it is dealing with. Book: Section 1.5.6 and Solution 1.4.

The Formula, Checked by the Machine

Listing 1.13 From `shelves_2d.cpp`: the measured offset of every `shelf[s][p]` against $(s \cdot \text{SLOTS} + p) \cdot \text{sizeof(Book)}$.

```
    for (int s = 0; s < SHELVES; s++) {
        for (int p = 0; p < SLOTS; p++) {
            const char* here = reinterpret_cast<const char*>(&shelf[s][p]);
            long actual = static_cast<long>(here - base);
            long formula = static_cast<long>((s * SLOTS + p) * sizeof(Book));
            std::printf("%-2d %-2d %-6s %-10ld %-24ld %s\n", s, p, shelf[s][p].id,
                        actual, formula, actual == formula ? "yes" : "NO");
        }
    }
```

```
sizeof(Book) = 196, SLOTS = 4
s  p  id     offset     (s*SLOTS+p)*sizeof(Book) same?
0  0  B1001  0          0                        yes
0  1  B1002  196        196                      yes
0  2  B1003  392        392                      yes
0  3  B1004  588        588                      yes
1  0  B1005  784        784                      yes
1  1  B1006  980        980                      yes
1  2  B1007  1176       1176                     yes
```

Listing 1.13 compares each measured offset with Equation (1.3). `shelf[1][2]` holds B1007: $(1 \cdot 4 + 2) \cdot 196 = 1176$, exactly the measured offset. A Java `Book[][]` or a Python list of lists is an array of row arrays; the rows may even differ in length (jagged). Book: Section 1.5.6.

Tables of Numbers

Listing 1.14 From `loan_table.cpp`: row totals, column totals and a grand total with two nested loops.

```
    int columnTotal[COLS] = {0};          // every element starts at 0
    int grandTotal = 0;
    for (int r = 0; r < ROWS; r++) {
        int rowTotal = 0;
        cout << left << setw(8) << shelfName[r];
        for (int c = 0; c < COLS; c++) {
            cout << right << setw(5) << loans[r][c];
            rowTotal += loans[r][c];
            columnTotal[c] += loans[r][c];
        }
        cout << setw(7) << rowTotal << '\n';
        grandTotal += rowTotal;
    }
```

```
shelf      W1   W2   W3   W4  total
Fiksi      31   28   35   40    134
Sejarah    12   15    9   14     50
Sains      20   22   18   25     85
total      63   65   62   79    269
```

In Listing 1.14 the outer loop walks the rows, the inner loop the columns of one row. `columnTotal` needs one running total per column, so it is an array set to zero by `= {0}`. The loops go row by row, in the order the elements lie in memory; that suits the processor's 64-byte cache lines. Book: Section 1.5.7.

A Python Trap: One Row, Three Times

`*` on a list repeats *references*: `[[0] * 4] * 3` is three references to the **same** row (Listing 1.15).

Listing 1.15 `aliasing_rows.py`: `[[0] * 4] * 3` makes three references to one row; a list comprehension makes three rows.

```python
# aliasing_rows.py - why [[0] * 4] * 3 is not a 3 x 4 table
wrong = [[0] * 4] * 3                    # three references to ONE row
right = [[0] * 4 for _ in range(3)]      # three separate rows

wrong[0][1] = 9
right[0][1] = 9
print("wrong:", wrong)
print("right:", right)
print("wrong rows are one object:", wrong[0] is wrong[1] is wrong[2])
```

```
wrong: [[0, 9, 0, 0], [0, 9, 0, 0], [0, 9, 0, 0]]
right: [[0, 9, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
wrong rows are one object: True
```

The correct idiom, `[[0] * 4 for _ in range(3)]`, runs `[0] * 4` three times and makes three separate rows. Book: Section 1.5.7.

`std::array` and `std::vector`

Raw arrays forget their length in a function, cannot be copied with `=`, and have a size fixed at compile time.

`std::array<T, N>` knows `size()`, copies and compares, offers `at(i)`, and costs nothing extra.

`std::vector<T>` grows: `push_back`, `pop_back`, `size()`, `v[i]`, `v.at(i)`. It keeps a heap block, a **size** and a **capacity**.

When the block is full, `push_back` allocates a larger block, copies every element, frees the old one, and only then adds the new element.

If you know roughly how many elements are coming, `v.reserve(n)` sets the capacity once and avoids every regrowth. Book: Section 1.5.8.

One Regrowth, Drawn

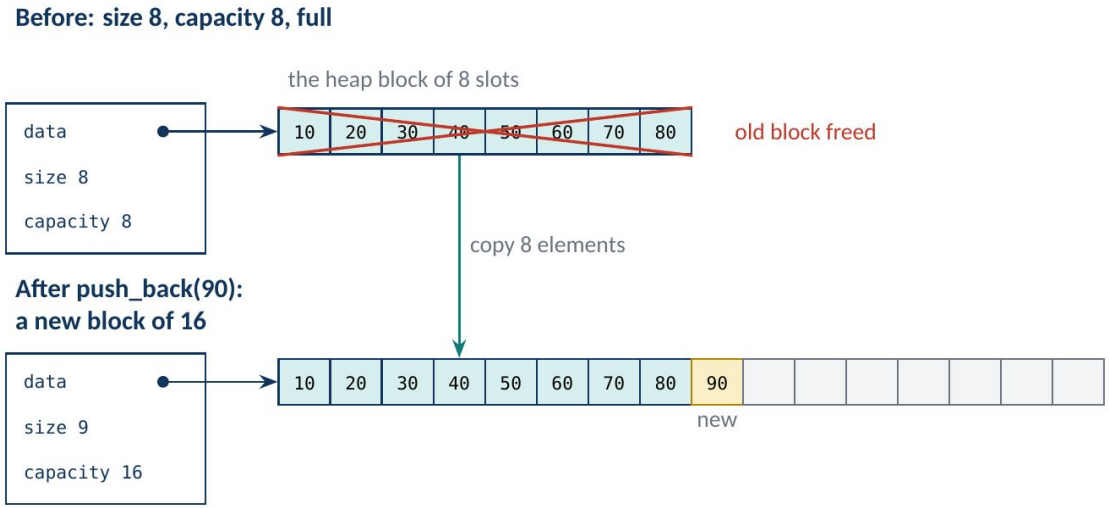


Figure 1.8 A full `std::vector` of capacity 8 receives a ninth element: a block of 16 is allocated, the 8 elements are copied, the old block is freed, and 90 is added.

Figure 1.8: a vector of capacity 8 receives a ninth element; a block of 16 is allocated, 8 elements are copied, the old block is freed, and 90 is added. Book: Section 1.5.8.

Worked Example 3: Watching a Vector Grow

Listing 1.16 From `vector_growth.cpp`: 100 `push_back`s, printing a line whenever the capacity changes.

```
    int moves = 0;                          // elements copied during regrowth
    for (int i = 1; i <= 100; i++) {
        size_t sizeBefore = v.size();
        v.push_back(i);
        if (v.capacity() != lastCapacity) {
            moves += static_cast<int>(sizeBefore);
            cout << "push_back #" << i << ": capacity " << lastCapacity
                 << " -> " << v.capacity() << " (moved " << sizeBefore << ")\n";
            lastCapacity = v.capacity();
        }
    }
```

```
push_back #1: capacity 0 -> 1 (moved 0)
push_back #2: capacity 1 -> 2 (moved 1)
push_back #3: capacity 2 -> 4 (moved 2)
push_back #5: capacity 4 -> 8 (moved 4)
push_back #9: capacity 8 -> 16 (moved 8)
push_back #17: capacity 16 -> 32 (moved 16)
push_back #33: capacity 32 -> 64 (moved 32)
push_back #65: capacity 64 -> 128 (moved 64)
```

With GCC the capacity doubles: 1, 2, 4, ..., 128. Microsoft's library multiplies by 1.5; the argument on the next page works for any constant factor. Book: Section 1.5.8.

Why Doubling Is Cheap

To reach n elements, the vector grew at sizes $1, 2, 4, \ldots$ up to the largest power $2^k < n$, copying that many each time:

$$1 + 2 + 4 + \cdots + 2^k = 2^{k+1} - 1 < 2n \tag{1.5}$$

For $n = 100$: $1 + 2 + 4 + 8 + 16 + 32 + 64 = 127 < 200$, exactly the count in Listing 1.16.

Growing by one slot at a time would copy $n(n-1)/2$: nearly 5,000 for $n = 100$, five billion for $n = 100,000$.

Equation (1.5): over a long run, fewer than two copies per `push_back`, a constant. Java's `ArrayList` grows by 1.5 from a first capacity of 10 (202 copies for 100 elements, below $3n$). Book: Sections 1.5.8 and 1.5.9.

Python Lists and the `array` Module

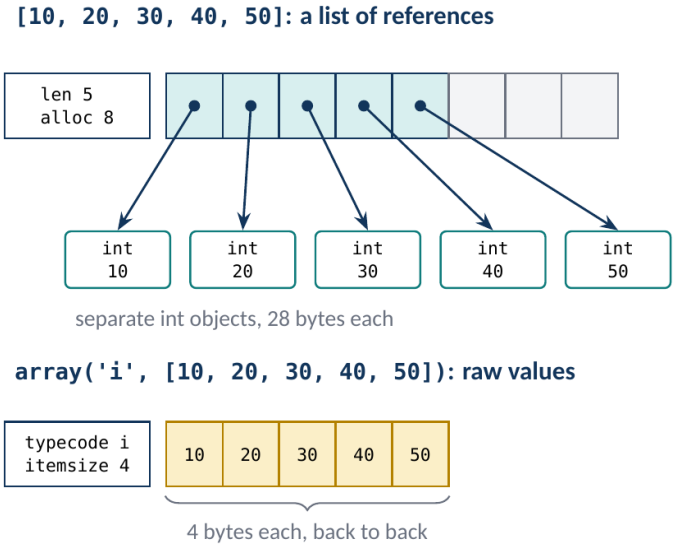


Figure 1.9 A Python `list` stores references to separate objects; an `array('i')` from the `array` module stores the raw 4-byte values themselves, back to back.

Figure 1.9: a list holds eight-byte references to 28-byte integer objects; `array('i')` packs raw 4-byte values. Java's `ArrayList<Integer>` also holds references. Book: Sections 1.5.9 and 1.5.10.

Insertion: Making Room

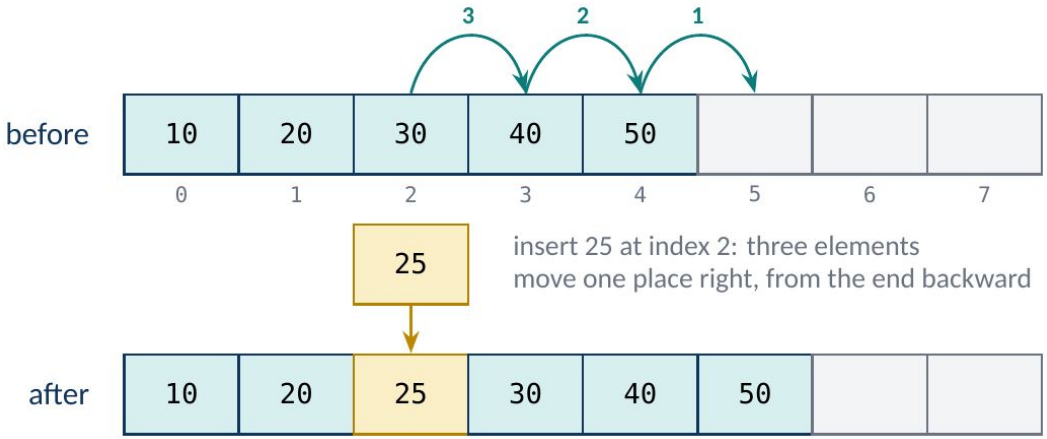


Figure 1.10 Inserting 25 at index 2: the elements at indexes 4, 3 and 2 move one place right, in that order, and 25 is written into the freed slot.

Figure 1.10: to insert 25 at index 2, move 50, then 40, then 30 one place right, from the end backward; moving from the front would overwrite 40 before it had been saved. Book: Section 1.6.3.

Insertion and Deletion, Precisely

Pseudocode 1.1 `ARRAY-INSERT`: open a gap at index `pos` by moving the elements after it one place right, then write `x` into the gap.

```
ARRAY-INSERT(A, n, pos, x):          // A[0..n-1] holds n elements; 0 ≤ pos ≤ n
 1.  for i ← n downto pos + 1         // walk from the end back to pos
 2.      A[i] ← A[i - 1]              // move one element one place right
 3.  A[pos] ← x                       // write the new element into the gap
 4.  n ← n + 1
```

Pseudocode 1.2 `ARRAY-DELETE`: close the gap left by the element at index `pos` by moving the elements after it one place left.

```
ARRAY-DELETE(A, n, pos):             // A[0..n-1] holds n elements; 0 ≤ pos ≤ n - 1
 1.  for i ← pos to n - 2
 2.      A[i] ← A[i + 1]              // move one element one place left
 3.  n ← n - 1
```

Pseudocode 1.1 walks from the end back to `pos`; Pseudocode 1.2 is its mirror image, moving elements left from the gap forward. Book: Section 1.6.3.

Counting the Moves

Line 2 of Pseudocode 1.1 runs once for each i from n down to $pos+1$, as Equation (1.6) states:

$$M_{\text{insert}}(n, pos) = n - pos \tag{1.6}$$

At the end, $pos = n$: no moves. At the front, $pos = 0$: all n. Averaged over the $n+1$ positions, Equation (1.7):

$$\frac{1}{n+1} \sum_{pos=0}^{n} (n - pos) = \frac{1}{n+1} \cdot \frac{n(n+1)}{2} = \frac{n}{2} \tag{1.7}$$

Write $0+1+\cdots+n$ forward and backward and add column by column: $n+1$ columns of n, so twice the sum is $n(n+1)$. Deletion moves $n-1-pos$ elements, $(n-1)/2$ on average. Book: Section 1.6.3.

Worked Example 4: Insertion, Counted

Listing 1.17 From `insert_shift.cpp`: insertion by shifting, returning the number of moves.

```cpp
// Inserts x at index pos of a[0..n-1] (capacity permitting).
// Returns the number of elements moved one place to the right.
int insertAt(vector<int>& a, int& n, int pos, int x) {
    int moves = 0;
    for (int i = n; i > pos; i--) {       // walk from the end back to pos
        a[i] = a[i - 1];
        moves++;
    }
    a[pos] = x;
    n++;
    return moves;
}
```

```
moves needed to insert one element into an array of n:
     n   front  middle   end
    10      10       5     0
   100     100      50     0
  1000    1000     500     0
 10000   10000    5000     0
```

Listing 1.17 is Pseudocode 1.1 with a counter, run on arrays of 10 to 10,000 elements. The table is Equation (1.6) in action: multiply the size by ten and the work at the front or in the middle is multiplied by ten; only the end is free. `v.insert`, `list.add(pos, x)` and `lst.insert(pos, x)` do the same shifting behind one line. Book: Section 1.6.3.

Big-O, Informally

DEFINITION. We write $f(n)=O(g(n))$ if there are positive constants c and n_0 such that Equation (1.8) holds: from some size onward, f grows no faster than a constant multiple of g.

$$f(n)\le c\cdot g(n)\text{ for all }n\ge n_0 \tag{1.8}$$

It ignores **constant factors** ($n/2$ and n are both $O(n)$) and **small inputs**; what remains is the *shape* of the growth.

$n/2\le 1\cdot n$ for $n\ge 1$, so $n/2=O(n)$. The sum loop takes about $3n+2\le 5n$ steps for $n\ge 1$: $O(n)$.

Two shapes today: $O(1)$, constant, and $O(n)$, linear — double n and the work doubles. Chapter 2 defines the notation with full rigor. Book: Section 1.6.4.

The Cost of Array Operations

Table 1.4 The cost of the basic operations on an array of n elements.

Operation	Steps	Big-O	Why
Read or write `A[i]`	1	$O(1)$	Equation (1.2): one computation of the address
Insert or delete at the end (room available)	1	$O(1)$	nothing needs to move
Insert at index `pos`	$n - pos$	$O(n)$	Equation (1.6); $n/2$ on average
Delete at index `pos`	$n - 1 - pos$	$O(n)$	the mirror of insertion
Traverse, sum, maximum, count, reverse	about n	$O(n)$	every element is visited once
Search for a value (unsorted)	up to n	$O(n)$	may need to look at every element
`push_back` / `add` / `append`, averaged over many	fewer than 2 or 3 copies each	$O(1)$ amortized	Equation (1.5)

Table 1.4 is the first of many such tables; every data structure is a choice of which rows to make cheap. The last row is ***amortized*** $O(1)$*: a guarantee about every long sequence of operations, not "usually fast". Book: Sections 1.6.4 and 1.6.5.*

Your First Program

Listing 1.18 `hello.cpp`: the first program; it proves that the compiler, the editor and the terminal work together.

```cpp
// hello.cpp - the first program of the book
#include <iostream>
#include <string>
using namespace std;

int main() {
    string name;
    cout << "What is your name? ";
    if (!getline(cin, name) || name.empty()) {
        name = "friend";                  // no input: greet a friend
    }
    cout << "Hello, " << name << "! Welcome to Data Structure.\n";
    cout << "Language version: C++ " << __cplusplus << '\n';
    return 0;
}
```

```
What is your name? Sari
Hello, Sari! Welcome to Data Structure.
Language version: C++ 201703
```

Workbench: VS Code with the C/C++, Java and Python extensions, GCC (MSYS2 UCRT64 on Windows), JDK 25, Python 3.14. Listing 1.18 compiles with `g++ -std=c++17 -O2 -Wall -Wextra -o hello hello.cpp`; `201703` *is* `__cplusplus` *under C++17. Book: Sections 1.7.7 and 1.7.8.*

Profitina in Practice

A fixed row of scores per page: a site scan scores each page against a fixed list of signals; slot k always means signal k, read in $O(1)$.

A table of pages by signals: row totals say how one page does, column totals how the site does on one signal — the shape of the loan table.

A history that grows only at the end: the weekly score series is a growable array; appending is amortized $O(1)$, and nothing is ever inserted in the middle.

Fail loudly: when the local AI model behind the Content Engine is unavailable, the application says so — the principle of `at()` at the smallest scale.

Newest-first feeds append at the end and read backward, instead of inserting at index 0 and paying n moves each time. Book: Section 1.8.

Common Mistakes

`i <= n` or starting at 1: visits the non-existent `a[n]` and skips `a[0]`. The condition is always `i < n`.

An uninitialized array, or **`sizeof` inside a function** to find the length: leftover bytes, and `8/4=2` whatever the real length.

Losing a value while swapping, or **shifting in the wrong direction**: use a temporary; insert from the end backward.

Rows and columns mixed up; a Java or Python "2-D array" taken for one block; `[[0] * c] * r` in Python.

A reference copied and called a copy; inserting at the front in a loop ($n^2/2$ moves); overflow and integer division; a forgotten `delete`.

Each mistake in the book comes with its fix. Book: Section 1.9.

Summary: Five Things to Remember

1. A data structure is an organization of data plus its operations; the right choice changes n steps into $\log_2 n$ or 1 (Table 1.1).

2. Memory is a row of addressed bytes; a type fixes how many bytes and how they are read; locals live on the stack, `new` on the heap.

3. Element i of an array is at $b + i \cdot s$ (Equation (1.2)), so access is $O(1)$; a C++ table is row-major (Equation (1.3)).

4. Insertion or deletion in the middle shifts $n - pos$ elements, $O(n)$; appending to a vector that doubles is amortized $O(1)$ (Equation (1.5)).

5. C++ does not check bounds (`at()` does); Java and Python always check, and Python accepts negative indexes.

The workbench is VS Code with C++, Java and Python, checked with the verification commands of the book. Book: Section 1.10.

Next Time

Chapter 2 — Searching. Sequential search, the sentinel, binary search and interpolation search, and O-notation in full.

The contract "a position, or -1" of `findFirst` becomes the contract of every search.

Before then: install the workbench, run `hello` in all three languages, and work Exercises 1.3, 1.4, 1.7 and 1.9.

Pustaka Ceria's catalog, sorted by code, is where binary search will halve its way to a book in seventeen looks among eighty thousand. Book: Sections 1.11 and 1.12.

Lecture 2 — Searching

Course: Data Structure (EF234201)

Book: Irfan Subakti, *Data Structure with C++, Java and Python*, Chapter 2

Lecturer: Irfan Subakti — yifana@gmail.com

Today we answer the oldest question a collection is ever asked — is it here, and where? — in six ways, and learn to choose between them with numbers rather than guesses. Book: Chapter 2, Sections 2.1–2.15.

What This Meeting Covers

1. **The problem and its unit of cost** — a position or -1; probes; best, worst and average case.
2. **Sequential search** — with and without a sentinel, and exactly what each costs.
3. **Binary search** — the trace, the invariant, the proof, $\lfloor \lg n \rfloor + 1$ probes, the recursive form.
4. **Off-by-one traps** — closed and half-open intervals, lower and upper bound, the overflowing middle.
5. **Interpolation, jump and two-dimensional search** — and what a real catalog of 10,000 books says about all of them.
6. **Libraries, the lower bound for comparison search, Profitina Laundry.**

One idea runs through the whole meeting: the more you know about how the data is arranged, the less of it you need to look at. Book: Section 2.1.

Where We Left Off

Chapter 1 gave us a workbench in three languages and a picture of memory: cell i of an array lives at the address of Equation (2.1).

$$address(A[i]) = base + i \cdot w, \tag{2.1}$$

So **any cell costs one step to reach**. Every search today is a strategy for deciding *which* cells to look at.

Pustaka Ceria's books now sit in an array. A borrower asks for a code: does the library have it, and where?

Searching is the most frequently performed operation in computing: every log-in, every barcode scan, every answer engine that quotes a page. Book: Section 2.1.

The Searching Problem

> **DEFINITION.** Given n records $A[0], \ldots, A[n-1]$ and a search key x, return an index i whose key equals x, or report that none exists. A search that finds the key is **successful**; one that does not is **unsuccessful**.

Return a **position**, never just FOUND: the caller needs it to print, lend or move the book. "Absent" is -1, never a valid index.

The unit of cost is the **probe**: one inspection of one element.

We ask for the **best**, the **worst** (the guarantee) and the **average** case, and summarize with $O(\cdot)$; $\lg n$ means $\log_2 n$.

$\lg 8 = 3$ because $8 \to 4 \to 2 \to 1$ takes three halvings. $O(\lg n)$ grows by one each time n doubles. Book: Sections 2.2.1 and 2.2.2.

Pustaka Ceria's Computing Shelf

Table 2.1 Pustaka Ceria's computing shelf, sorted by code. Sequential search will also meet these books in the order they arrived: 072, 025, 096, 041, 088, 034, 063, 056.

Index	Code	Title	Writer
0	025	The C++ Programming	James Wood
1	034	Mastering Delphi 6	Marcopolo
2	041	Professional C#	Simon Webe
3	056	Pure JavaScript v2	Michael Bolton
4	063	Advanced JSP & Servlet	David Dunn
5	072	Calculus Make it Easy	Gunner Christian
6	088	Visual Basic 2005 Express	Antonie
7	096	Artificial Life : Volume 1	Gloria Virginia

Table 2.1 *is the running example. Books arrive in* ***arrival order****, 072, 025, 096, 041, 088, 034, 063, 056; the sorted order exists only after someone has sorted them, the subject of Chapter 3. Which searches are possible depends on which order we have. Book: Section 2.2.3.*

Sequential Search

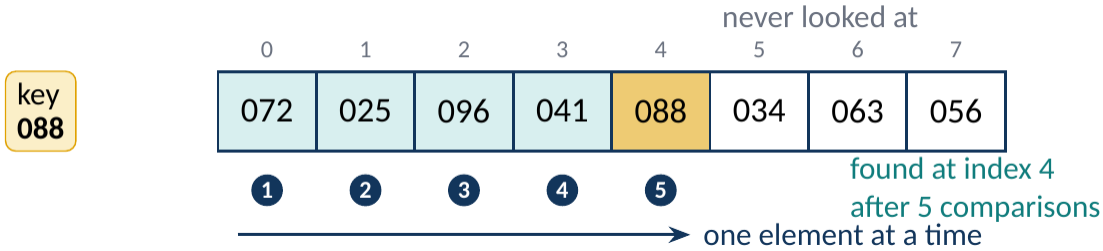


Figure 2.1 Sequential search for code 088 on the shelf in arrival order. The numbered circles give the order of the probes; the search stops at the fifth one, and the last three cells are never examined.

***Figure 2.1**: with no order to exploit, look at the first card, then the second, and stop at the match or at the end. It works on any array, which is its strength; it can skip nothing, which is its weakness. Book: Section 2.3.1.*

The Procedure, and Flag Versus Index

Pseudocode 2.1 `SEQUENTIAL-SEARCH`: examine the elements from the front and stop at the first match.

```
SEQUENTIAL-SEARCH(A, n, x):
 1.  for i ← 0 to n − 1
 2.      if A[i] = x
 3.          return i                  // found: stop at the first match
 4.  return −1                         // every element examined, none matched
```

The first version most people write sets a **flag** and runs to the end: always n comparisons, and it forgets *where* it found the key.

Pseudocode 2.1 cures both faults: return from inside the loop, and return the index.

Stop early for existence, never for counting: "how many times was 041 borrowed?" must look at every entry.

The loan log 041, 088, 025, 041, 063, 041, 096, 088 holds book 041 three times, at positions 0, 3 and 5. Book: Sections 2.3.1 and 2.3.2.

Sequential Search in C++

Listing 2.1 From `sequential_search.cpp`: early exit with the index, and the counting loop that must run to the end.

```cpp
// Stop at the first match; return its index, or -1 if key is absent.
int sequentialSearch(const vector<int>& data, int key, int& comparisons) {
    comparisons = 0;
    for (size_t i = 0; i < data.size(); i++) {
        comparisons++;
        if (data[i] == key) return (int)i;
    }
    return -1;
}

// Every position holding key: the loop must not stop early.
vector<int> countOccurrences(const vector<int>& data, int key) {
    vector<int> where;
    for (size_t i = 0; i < data.size(); i++)
        if (data[i] == key) where.push_back((int)i);
    return where;
}
```

Listing 2.1 *on* [8,10,6,−2,11,7,1,100]*: key 8 costs 1 comparison, 6 costs 3, 100 costs 8, and the absent 5 costs all 8. On the shelf in arrival order, code 088 costs 5, exactly as* ***Figure 2.1*** *predicted. Book: Section 2.3.3.*

How Much Does Sequential Search Cost?

Best case: the key is in $A[0]$: one probe, $O(1)$. **Worst case:** last or absent: n probes, $O(n)$.

Average, successful, every position equally likely, Equation (2.2):

$$C_{\text{succ}}(n) = \frac{1}{n}\sum_{i=0}^{n-1}(i+1) = \frac{1}{n}\cdot\frac{n(n+1)}{2} = \frac{n+1}{2}. \tag{2.2}$$

Average when the key is present with probability p, Equation (2.3):

$$C(n,p) = p\cdot\frac{n+1}{2} + (1-p)\cdot n. \tag{2.3}$$

Eight books: 4.5 probes on average; ten thousand: 5,000.5. At $p = 1/2$ about $3/4\,n$. Double the catalog and you double the average search: that is $O(n)$ in practice. Book: Section 2.3.4.

When Sequential Search Is the Right Tool

> **KEY IDEA.** Sequential search is the only search in this chapter that works on **unsorted** data, and on data you can only read from front to back — a file, a stream, a linked list. When the data is small, searched once, or cannot be sorted, it is the right tool. When the data is large and searched often, it is the tool to replace.

The sum $1 + 2 + \cdots + n = n(n + 1)/2$: write it forward and backward and add column by column.

An unsuccessful sequential search *always* costs exactly n probes.

Chapter 7 searches a linked list, where sequential search is the only choice. Book: Section 2.3.4.

The Sentinel: One Test Where Two Would Do

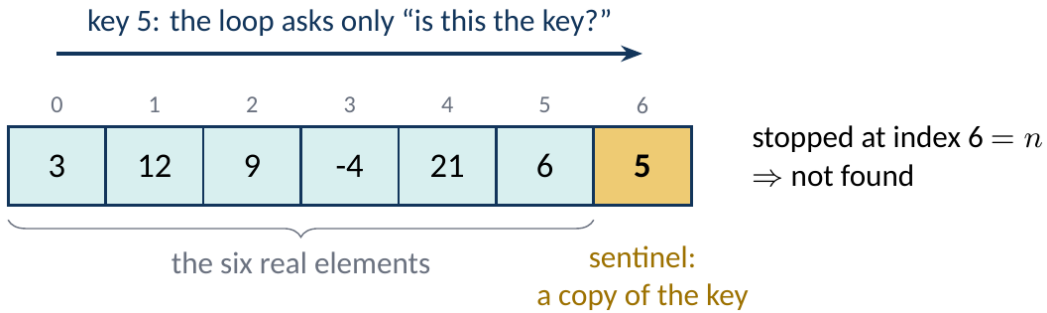


Figure 2.2 Sequential search with a sentinel. The key 5 is copied into the spare cell at index 6, so the loop must stop there at the latest; stopping at index $6 = n$ means the key is not among the real elements.

Figure 2.2*: every step of* ***Pseudocode 2.1*** *asks “is $i < n$?” and “is $A[i]$ the key?”. Plant a copy of the key in a spare cell at index n and the loop is sure to stop; afterwards, $i < n$ means found and $i = n$ means absent. Book: Section 2.4.1.*

SENTINEL-SEARCH

Pseudocode 2.2 `SENTINEL-SEARCH`: plant a copy of the key after the last element so that the loop needs only one test per step.

```
SENTINEL-SEARCH(A, n, x):          // A has a spare cell A[n]
 1.  saved ← A[n]
 2.  A[n] ← x                      // plant the sentinel
 3.  i ← 0
 4.  while A[i] ≠ x                // one test per step: no bound check
 5.      i ← i + 1
 6.  A[n] ← saved                  // put the spare cell back as it was
 7.  if i < n
 8.      return i
 9.  return -1
```

Pseudocode 2.2 *saves the spare cell first and restores it before returning, so the array is left as it was found. Book: Section 2.4.1.*

Counting the Tests

Listing 2.2 From `sentinel_search.cpp`: the sentinel loop, with its only test counted.

```cpp
// data has one spare slot at index n; the key is planted there first.
int sentinelSearch(int data[], int n, int key, int& keyTests) {
    int saved = data[n];
    data[n] = key;  // the sentinel: the loop is now sure to stop
    keyTests = 0;
    int i = 0;
    while (true) {
        keyTests++;
        if (!(data[i] != key)) break;
        i++;
    }
    data[n] = saved;  // leave the array as we found it
    return i < n ? i : -1;
}
```

```
key 21  plain:    index 4   10 loop tests (5 bound + 5 key)
        sentinel: index 4    5 loop tests (key only)
key  5  plain:    not found 13 loop tests (7 bound + 6 key)
        sentinel: not found  7 loop tests (key only)
```

***Listing 2.2**: for the absent 5 the plain loop makes $2n + 1 = 13$ tests, the sentinel loop $n + 1 = 7$. It halves the tests, not the probes, and both stay $O(n)$. It needs a writable spare cell and must never be used by two threads on one array. On the real catalog it made no difference in C++ or Java and saved about 9% in Python. Book: Sections 2.4.2 and 2.4.3.*

Binary Search: Using the Order

On the sorted shelf, open the middle: 056. Every book to its left has a smaller code, so 088 cannot be there. **Half the shelf is gone in one look.**

This is how you look up a word in a paper dictionary.

> **DEFINITION. Binary search** searches a *sorted* $A[0..n-1]$ by keeping an interval $A[low..high]$ that must contain x if x is present. It probes $A[mid]$: equal, found; smaller, the interval becomes $A[mid+1..high]$; larger, $A[low..mid-1]$. An empty interval ($low > high$) means absent.

Binary search is the most important algorithm of this chapter. Book: Section 2.5.1.

A Trace by Hand



Figure 2.3 Binary search for 17 in [3,9,11,12,15,17,23,31,35]. Grey cells have been ruled out; the amber cell is the one probed. Three probes find the key.

***Figure 2.3**: three probes find 17 in* [3,9,11,12,15,17,23,31,35]*; sequential search would need six. Book: Section 2.5.2.*

Worked Example 1: Search for 17, Then for 10

17: $low = 0$, $high = 8$, $mid = 4$, $A[4] = 15 < 17$, so $low \leftarrow 5$. Then $mid = 6$, $A[6] = 23 > 17$, so $high \leftarrow 5$. Then $mid = 5$: found.

10: $mid = 4, 15 > 10, high \leftarrow 3$; $mid = 1, 9 < 10, low \leftarrow 2$; $mid = 2, 11 > 10, high \leftarrow 1$.

Now $low = 2 > high = 1$: empty, 10 is absent — and $low = 2$ is exactly where 10 would be inserted.

```
iterative search for 17
  low=0 high=8 mid=4 data[mid]=15 -> 17 > 15, low = 5
  low=5 high=8 mid=6 data[mid]=23 -> 17 < 23, high = 5
  low=5 high=5 mid=5 data[mid]=17 -> found
  result: index 5 after 3 probes
iterative search for 10
  low=0 high=8 mid=4 data[mid]=15 -> 10 < 15, high = 3
  low=0 high=3 mid=1 data[mid]=9 -> 10 > 9, low = 2
  low=2 high=3 mid=2 data[mid]=11 -> 10 < 11, high = 1
  result: not found after 3 probes; it would go at index 2
```

The program prints the hand trace line for line. The insertion point left in low is the seed of lower bound, later today. Book: Sections 2.5.2 and 2.5.6.

BINARY-SEARCH

Pseudocode 2.3 `BINARY-SEARCH`, iterative, on the closed interval $[low, high]$.

```
BINARY-SEARCH(A, n, x):           // A[0..n-1] sorted in non-decreasing order
 1.  low ← 0
 2.  high ← n − 1
 3.  while low ≤ high             // the interval A[low..high] is not empty
 4.      mid ← low + ⌊(high − low)/2⌋
 5.      if A[mid] = x
 6.          return mid
 7.      else if A[mid] < x
 8.          low ← mid + 1        // x can only be to the right of mid
 9.      else
10.          high ← mid − 1       // x can only be to the left of mid
11.  return −1                    // interval empty: x is absent
```

Pseudocode 2.3 *works on the closed interval $[low, high]$. Line 4 computes the middle as $low + \lfloor (high - low)/2 \rfloor$, for a reason we meet shortly. The first binary search was published in 1946; the first correct for every n, in 1962. Book: Section 2.5.3.*

The Loop Invariant

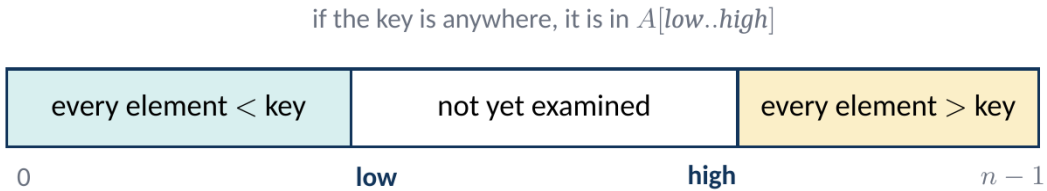


Figure 2.4 The loop invariant of binary search. Everything left of *low* is known to be smaller than the key, everything right of *high* is known to be larger, and the key, if present, is in the middle region.

***Figure 2.4**: at the start of every iteration, everything in $A[0..low-1]$ is less than x and everything in $A[high+1..n-1]$ is greater; so if x is present, it is in $A[low..high]$. Book: Section 2.5.3.*

Proving Binary Search Correct

Initialization. $low = 0$, $high = n - 1$: both outer regions are empty, and a statement about every element of an empty region is true.

Maintenance. If $A[mid] < x$, the sorted order makes all of $A[low..mid]$ less than x, so $low \leftarrow mid + 1$ keeps the invariant. The other case is the mirror image.

Termination. Either line 6 returns a true match, or $low > high$: the middle region is empty, x is absent, and low is the insertion point.

The loop ends. $high - low + 1$ is a non-negative integer that shrinks on every iteration that does not return.

The defense against binary-search bugs is not cleverness but an invariant. Only about one professional programmer in ten wrote a correct binary search in Bentley's classes. Book: Section 2.5.3.

How Many Probes?

An interval of m elements leaves at most $\lfloor m/2 \rfloor$ after one probe, which gives the recurrence of Equation (2.4):

$$T(0) = 0, \qquad T(m) = 1 + T(\lfloor m/2 \rfloor) \quad (m \geq 1). \tag{2.4}$$

Unfolding it: sizes $n, \lfloor n/2 \rfloor, \lfloor n/4 \rfloor, \ldots$, while $2^k \leq n$, so Equation (2.5):

$$T(n) = \lfloor \lg n \rfloor + 1. \tag{2.5}$$

EXAMPLE. $n = 9$: $\lfloor 3.17 \rfloor + 1 = 4$ probes at most. $n = 10^4$: 14. A million: 20. A billion: 30 — against a billion for sequential search.

Doubling n adds one probe to binary search and doubles sequential search. A successful search averages about $\lg n - 1$ probes; on 10^4 keys we measured 12.4. Book: Section 2.5.4.

Binary Search, Recursively

Pseudocode 2.4 `BINARY-SEARCH-REC`: the recursive form. The first call is `BINARY-SEARCH-REC(A, x, 0, n − 1)`.

```
BINARY-SEARCH-REC(A, x, low, high):
 1.  if low > high
 2.      return −1                          // empty interval: absent
 3.  mid ← low + ⌊(high − low)/2⌋
 4.  if A[mid] = x
 5.      return mid
 6.  else if A[mid] < x
 7.      return BINARY-SEARCH-REC(A, x, mid + 1, high)
 8.  else
 9.      return BINARY-SEARCH-REC(A, x, low, mid − 1)
```

Pseudocode 2.4*: correct for this interval if correct for every shorter one — proof by induction, the theme of Chapter 10. At most* $\lfloor \lg n \rfloor + 2$ *stack frames; the tail call converts mechanically into the loop. Book: Sections 2.5.5 and 2.5.6.*

Recursive Calls, Counted

Listing 2.3 From `binary_search.cpp`: the recursive form, counting calls.

```cpp
// Same search, written as a function that calls itself on one half.
int binarySearchRec(const vector<int>& data, int key, int low, int high,
                    int& calls) {
    calls++;
    if (low > high) return -1;
    int mid = low + (high - low) / 2;
    if (data[mid] == key) return mid;
    if (data[mid] < key) return binarySearchRec(data, key, mid + 1, high, calls);
    return binarySearchRec(data, key, low, mid - 1, calls);
}
```

```
recursive search for 17: index  5, 3 calls
recursive search for 10: index -1, 4 calls
recursive search for  3: index  0, 3 calls
recursive search for 35: index  8, 4 calls
recursive search for 40: index -1, 5 calls
```

***Listing 2.3**: an unsuccessful search makes one last call on an empty interval that probes nothing, so 10 needed 3 probes in 4 calls; 40 needed 4 probes, the maximum that Equation (2.5) promised for $n = 9$, in 5 calls. Book: Section 2.5.6.*

Closed and Half-Open Intervals

Table 2.2 The two interval conventions. Each row must be taken from the same column; a program that takes its loop condition from one column and its update from the other is wrong.

	Closed [low,high]	Half-open [low,high)
Initial values	$low = 0, high = n - 1$	$low = 0, high = n$
Interval is empty when	$low > high$	$low = high$
Loop condition	`low <= high`	`low < high`
Discard left part and mid	$low \leftarrow mid + 1$	$low \leftarrow mid + 1$
Discard mid and right part	$high \leftarrow mid - 1$	$high \leftarrow mid$
Number of elements	$high - low + 1$	$high - low$

***Table 2.2**: both conventions are correct; mixing them is not, and almost every binary-search bug is a mixture. Half-open has two small advantages: the count is $high - low$, and the empty array needs no special case. Book: Section 2.6.1.*

Four Traps

1. **The wrong loop condition.** Closed with `low < high` never probes the last element: searching [5] for 5 fails. Half-open with `low <= high` probes $A[n]$.
2. **An update that does not shrink.** `low = mid` on a two-element interval changes nothing: the loop runs forever.
3. **An overflowing middle.** `(low + high) / 2` in 32-bit arithmetic.
4. **Ignoring duplicates.** **Pseudocode 2.3** returns *one* copy of a repeated key, and which one depends on n.

Every one of these has appeared in published code; the maintenance step of our proof fails exactly at trap 2. Book: Section 2.6.2.

Lower Bound and Upper Bound

DEFINITION. For sorted $A[0..n-1]$, **lower bound** is the smallest i with $A[i] \geq x$, and **upper bound** the smallest i with $A[i] > x$ (either is n if none). The copies of x occupy $A[lower..upper-1]$; their number is $upper - lower$; if x is absent, lower bound is where it should be inserted.

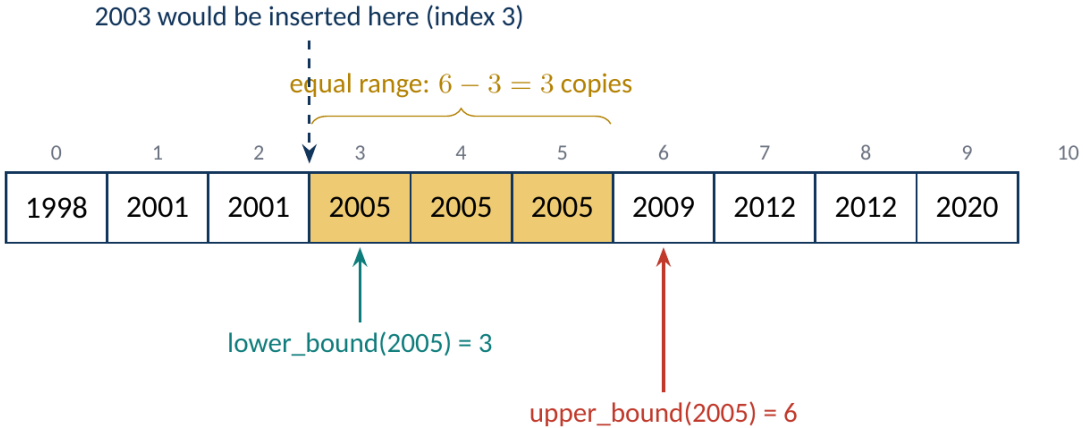


Figure 2.5 Lower bound and upper bound of 2005 in a sorted list of publication years. The three copies of 2005 occupy indices 3 to 5, the half-open range [3,6). For the absent 2003, lower bound and upper bound are both 3, the place where 2003 would be inserted.

***Figure 2.5**: the three copies of 2005 occupy* [3,6)*; for the absent 2003 both bounds are 3. Book: Section 2.6.3.*

LOWER-BOUND

Pseudocode 2.5 `LOWER-BOUND`: the first position whose element is not less than x. Changing `<` to `≤` in line 5 gives `UPPER-BOUND`.

```
LOWER-BOUND(A, n, x):              // A[0..n-1] sorted
 1.  low ← 0
 2.  high ← n                      // half-open interval [low, high)
 3.  while low < high
 4.      mid ← low + ⌊(high − low)/2⌋
 5.      if A[mid] < x
 6.          low ← mid + 1         // A[0..mid] are all < x
 7.      else
 8.          high ← mid            // A[mid..n-1] are all ≥ x
 9.  return low                    // first index with A[low] ≥ x (or n)
```

KEY IDEA. Lower and upper bound answer "is it here?" ($upper > lower$), "first copy?" ($lower$), "last?" ($upper - 1$), "how many?" ($upper - lower$), "where would it go?" ($lower$), and range counts $lower(b) - lower(a)$.

Pseudocode 2.5 *never tests for equality. Its invariant: $A[0..low-1]$ are all less than x and $A[high..n-1]$ are all at least x; when $low = high$ the unknown region is gone. At most $\lceil lg(n+1) \rceil$ probes, and nearly that many every time. Book: Section 2.6.3.*

Lower Bound in C++

Listing 2.4 From `bounds.cpp`: lower bound on the half-open interval; upper bound differs only in `<=`.

```cpp
// First index i with a[i] >= key (a.size() if none). Half-open [low, high).
int lowerBound(const vector<int>& a, int key) {
    int low = 0, high = (int)a.size();
    while (low < high) {
        int mid = low + (high - low) / 2;
        if (a[mid] < key)
            low = mid + 1;  // a[0..mid] are all < key
        else
            high = mid;  // a[mid..] are all >= key
    }
    return low;
}
```

```
years: 1998 2001 2001 2005 2005 2005 2009 2012 2012 2020
key   lower upper count  meaning
2005      3     6     3  first at 3, last at 5
2001      1     3     2  first at 1, last at 2
2003      3     3     0  absent, insert at 3
1990      0     0     0  absent, insert at 0
2024     10    10     0  absent, insert at 10
```

Listing 2.4 *is the binary search to reach for first. Book: Sections 2.6.3 and 2.6.5.*

The Overflowing Middle

In 2006 Joshua Bloch reported that `int mid = (low + high) / 2;` in Java's library had been wrong for about two decades.

With $low + high > 2^{31} - 1 = 2{,}147{,}483{,}647$, the sum wraps to a negative number, and so does the index.

The repair, Equation (2.6), never forms a number larger than $high$:

$$mid = low + \left\lfloor \frac{high - low}{2} \right\rfloor, \tag{2.6}$$

```
low + high = 3500000000, but a 32-bit int holds -794967296
(low + high) / 2 in 32 bits = -397483648  <- a negative index
low + (high - low) / 2      = 1750000000
```

In C++ signed overflow is undefined behavior, so `bounds.cpp` computes in 64 bits what a 32-bit `int` would hold; Java overflows a real `int`; Python's integers never overflow, but the habit costs nothing. Book: Section 2.6.4.

Interpolation Search: Guess Where It Should Be

Looking up *Tanaka* in a telephone directory, you open it three quarters of the way through: you use the **values**, not only the order.

If the keys rise in a straight line from $A[low]$ to $A[high]$, the key sits at the position of Equation (2.7):

$$pos = low + \left\lfloor \frac{x - A[low]}{A[high] - A[low]} \cdot (high - low) \right\rfloor. \tag{2.7}$$

$x = A[low]$ gives low; $x = A[high]$ gives $high$. Binary search always uses the fraction $1/2$.

The idea goes back to Peterson in 1957. Book: Section 2.7.1.

The Shelf Is Almost a Straight Line

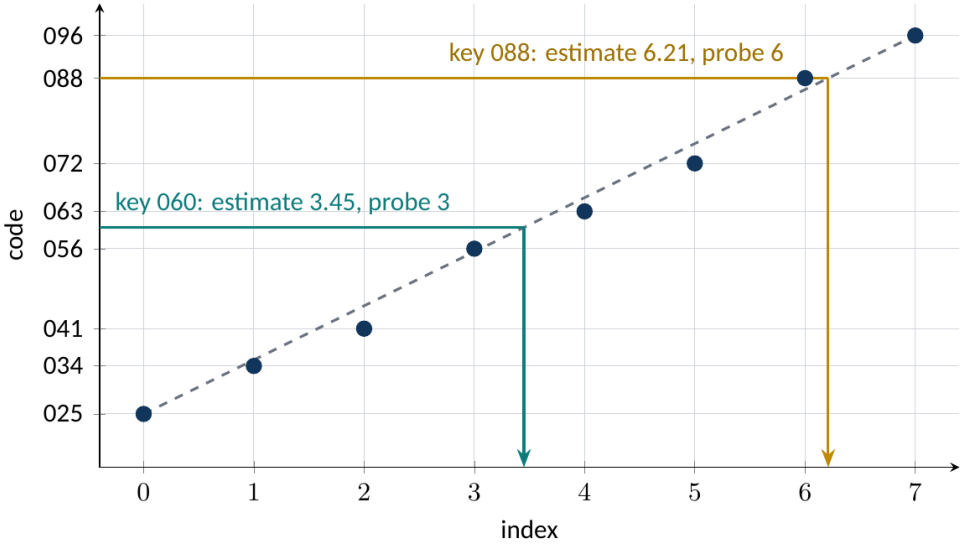


Figure 2.6 Interpolation search on the computing shelf. The dashed line runs straight from code[0] = 025 to code[7] = 096. For the key 088 it estimates position 6.21, and the probe at index 6 finds 088 at once; for the key 060 it estimates 3.45 and probes index 3.

***Figure 2.6**: the eight codes lie close to the dashed line from (0,025) to (7,096) — exactly the assumption of Equation (2.7). Book: Section 2.7.2.*

Worked Example 2: Interpolating by Hand

Key 088, $low = 0, high = 7$:

$$pos = 0 + \left\lfloor \frac{88 - 25}{96 - 25} \cdot 7 \right\rfloor = \left\lfloor \frac{63}{71} \cdot 7 \right\rfloor = \lfloor 6.21 \rfloor = 6$$

$A[6] = 088$: found with **one** probe; binary search needs three.

Key 060: $\lfloor (35/71) \cdot 7 \rfloor = \lfloor 3.45 \rfloor = 3$; $A[3] = 056 < 060$, so $low \leftarrow 4$. Now $060 < A[4] = 063$: outside the interval, absent after one probe.

The range check $A[low] \leq x \leq A[high]$ keeps the estimate inside the array; a second guard avoids dividing by zero when $A[low] = A[high]$. Book: Section 2.7.2.

INTERPOLATION-SEARCH

Pseudocode 2.6 `INTERPOLATION-SEARCH`: binary search with the middle replaced by an estimate, guarded against keys outside the range and against a zero denominator.

```
INTERPOLATION-SEARCH(A, n, x):    // A[0..n−1] sorted, numeric keys
 1.  low ← 0
 2.  high ← n − 1
 3.  while low ≤ high and A[low] ≤ x ≤ A[high]
 4.      if A[high] = A[low]
 5.          pos ← low                          // no slope: avoid dividing by 0
 6.      else
 7.          pos ← low + ⌊(x − A[low]) / (A[high] − A[low]) · (high − low)⌋
 8.      if A[pos] = x
 9.          return pos
10.      else if A[pos] < x
11.          low ← pos + 1
12.      else
13.          high ← pos − 1
14.  return −1
```

***Pseudocode 2.6**: on uniformly spread keys it averages about lglgn probes (4.26 on a million keys, against 18.8 for binary search); on bunched keys it degrades to $O(n)$. Every step costs a division, and the keys must be numbers. Book: Sections 2.7.2–2.7.4.*

Jump Search: Leaping Ahead in Blocks

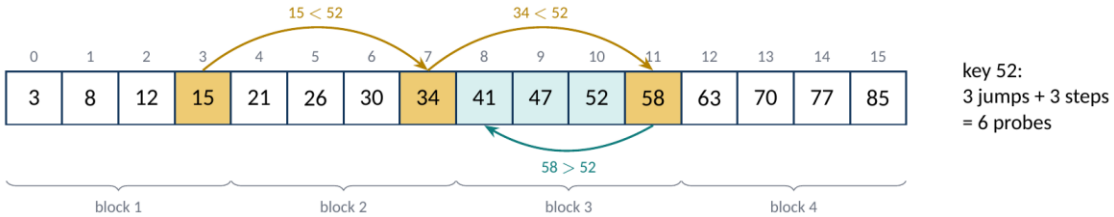


Figure 2.7 Jump search for 52 with blocks of four. The jumps probe the last element of each block (15, 34, 58); since $58 > 52$ the key must be in block 3, and a walk through 41, 47, 52 finds it.

***Figure 2.7**: jump m places at a time, probing the last element of each block (15, 34, 58); since $58 > 52$ the key is in block 3, and a short walk through 41, 47, 52 finds it. Useful where moving backward is expensive: tape, large file blocks. Book: Section 2.8.1.*

JUMP-SEARCH

Pseudocode 2.7 `JUMP-SEARCH`: jump over whole blocks, then walk through the single block that can contain the key.

```
JUMP-SEARCH(A, n, x, m):          // A[0..n−1] sorted, block size m
 1.  start ← 0
 2.  end ← min(m, n)              // current block is A[start..end−1]
 3.  while A[end − 1] < x         // jump while the whole block is too small
 4.      start ← end
 5.      end ← min(end + m, n)
 6.      if start ≥ n
 7.          return −1            // x is larger than every element
 8.  for i ← start to end − 1     // walk through the one block
 9.      if A[i] = x
10.          return i
11.      if A[i] > x
12.          return −1            // passed the place where x would be
13.  return −1
```

Pseudocode 2.7*: the programs also stop at once when a block's last element equals the key, and the walk skips that element. Book: Section 2.8.1.*

Choosing the Block Size

About n/m jumps and then up to $m - 1$ steps give the worst case of Equation (2.8):

$$f(m) = \frac{n}{m} + m - 1. \tag{2.8}$$

$f'(m) = -n/m^2 + 1 = 0$ gives the balance of Equation (2.9):

$$m^* = \sqrt{n}, \qquad f(m^*) = 2\sqrt{n} - 1. \tag{2.9}$$

```
key 90: not found  jump 4 + walk 0 = 4 probes
worst case n/m + m - 1 for the best m = floor(sqrt(n)):
  n =      16  m =    4  worst =    7 probes
  n =     100  m =   10  worst =   19 probes
  n =   10000  m =  100  worst =  199 probes
```

Without calculus: n/m and m have a fixed product n, and two positive numbers with a fixed product have the smallest sum when they are equal. Far better than n, far worse than $\lfloor \lg n \rfloor + 1$. Book: Sections 2.8.2 and 2.8.3.

Two-Dimensional Arrays in Memory

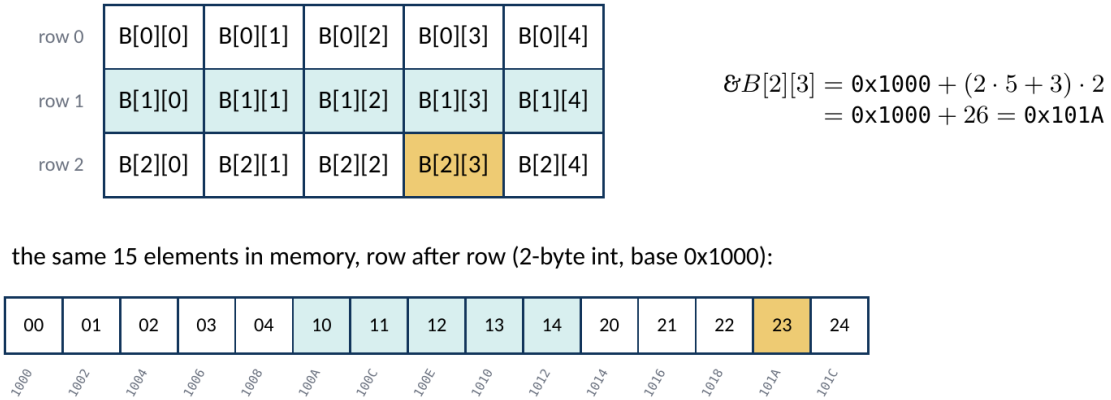


Figure 2.8 A 3 × 5 array and its row-major layout in memory. Each memory cell is labeled with the row and column of the element it holds; the small numbers are the addresses when an `int` occupies 2 bytes and the array starts at 0x1000.

$$address(B[i][j]) = base + (i \cdot C + j) \cdot w, \tag{2.10}$$

***Figure 2.8**: C and C++ store a grid row by row. Equation (2.10) turns $B[i][j]$ into element number $i \cdot C + j$ and then applies Equation (2.1). Fortran, MATLAB and R use column-major order. Book: Section 2.9.1.*

Worked Example 3: Two Addresses

`int A[10]` at 0x1000, an `int` of 2 bytes: $\&A[7] = 0\text{x}1000 + 7 \cdot 2 = 0\text{x}1000 + 14 = 0\text{x}100\text{E}$.

`int B[3][5]` at 0x1000: element number $2 \cdot 5 + 3 = 13$, byte offset $13 \cdot 2 = 26 = 0\text{x}1\text{A}$, so $\&B[2][3] = 0\text{x}101\text{A}$.

With today's 4-byte `int`: $0\text{x}1000 + 28 = 0\text{x}101\text{C}$ and $0\text{x}1000 + 52 = 0\text{x}1034$.

> **COMMON SLIP.** Adding the element *count* instead of the byte offset, $0\text{x}1000 + 7$, forgets the factor w.

`search_2d.cpp` prints exactly these four addresses. Book: Section 2.9.1.

Three Ways to Search a Grid

Unsorted grid: scan row by row, $O(RC)$ probes, in memory order.

Rows and columns both sorted: the **staircase**, $O(R + C)$, **Pseudocode 2.8**.

Whole grid sorted in row-major order: one binary search over $0..RC - 1$, with $r = \lfloor k/C \rfloor$ and $c = k \bmod C$.

Pseudocode 2.8 `STAIRCASE-SEARCH`: in a grid whose rows and columns both ascend, every probe rules out a whole row or a whole column.

```
STAIRCASE-SEARCH(M, R, C, x):      // rows and columns both ascending
 1.  r ← 0
 2.  c ← C − 1                     // start at the top-right corner
 3.  while r < R and c ≥ 0
 4.      if M[r][c] = x
 5.          return (r, c)
 6.      else if M[r][c] > x
 7.          c ← c − 1             // the rest of column c is even larger
 8.      else
 9.          r ← r + 1             // the rest of row r is even smaller
10.  return (−1, −1)
```

Start at the top-right corner: a larger element rules out the rest of its column, a smaller one the rest of its row. Book: Section 2.9.2.

The Staircase, Traced

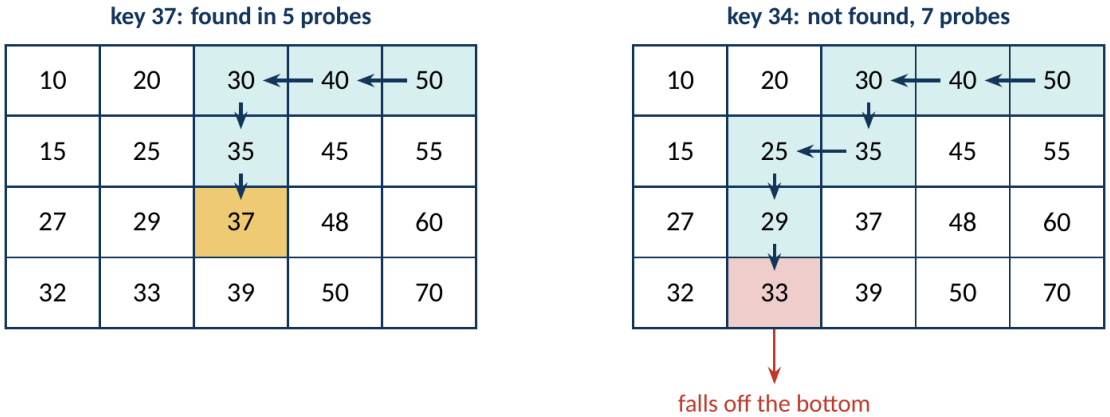


Figure 2.9 The staircase search on a 4 × 5 grid whose rows and columns ascend. For 37 the path 50, 40, 30, 35, 37 finds the key in five probes; for 34 the path steps left and down until it leaves the grid below 33.

***Figure 2.9**: for 37 the path 50, 40, 30, 35, 37 finds the key in five probes; for 34 the path steps left and down until it leaves the grid below 33, after seven. Book: Sections 2.9.2 and 2.9.3.*

Measuring on a Real Catalog

Table 2.3 Measured cost of the five searches on the 10,000-book catalog: average probes per search, and nanoseconds per search averaged over all 1,905 queries (best of nine runs).

Search	Probes, present	Probes, absent	Ns, C++	Ns, Java	Ns, Python
Sequential (file order)	4996.0	10000.0	2243	1481	135376
Sentinel (file order)	4996.0	10001.0	2248	1487	123270
Binary (sorted)	12.4	13.3	89	74	1054
Interpolation (sorted)	211.2	191.8	2612	3414	39667
Jump, $m = 100$ (sorted)	99.2	105.6	130	214	7782

Table 2.3*: the goodbooks-10k catalog of ten thousand real books; 1,000 present and 905 absent identifiers, best of nine runs. The probe counts match the theory: 4,996 and 10,000 for sequential search, 12.4 and 13.3 for binary search, never more than 14. Book: Sections 2.10.1 and 2.10.2.*

Probes Are Not Time

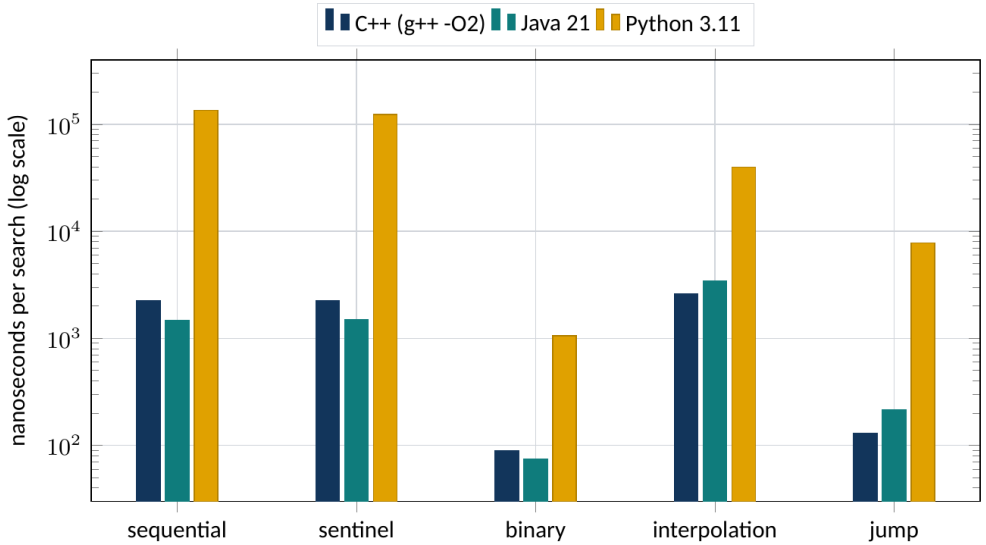


Figure 2.10 Nanoseconds per search on the 10,000-book catalog, by search method and language, on a logarithmic scale, measured in September 2026.

***Figure 2.10**: binary search was about 25 times faster than sequential in C++, not 600, because scans read memory in order. Interpolation failed on bunched identifiers (211 probes). Binary search in Python beat sequential search in C++. Book: Section 2.10.3.*

How Probe Counts Grow

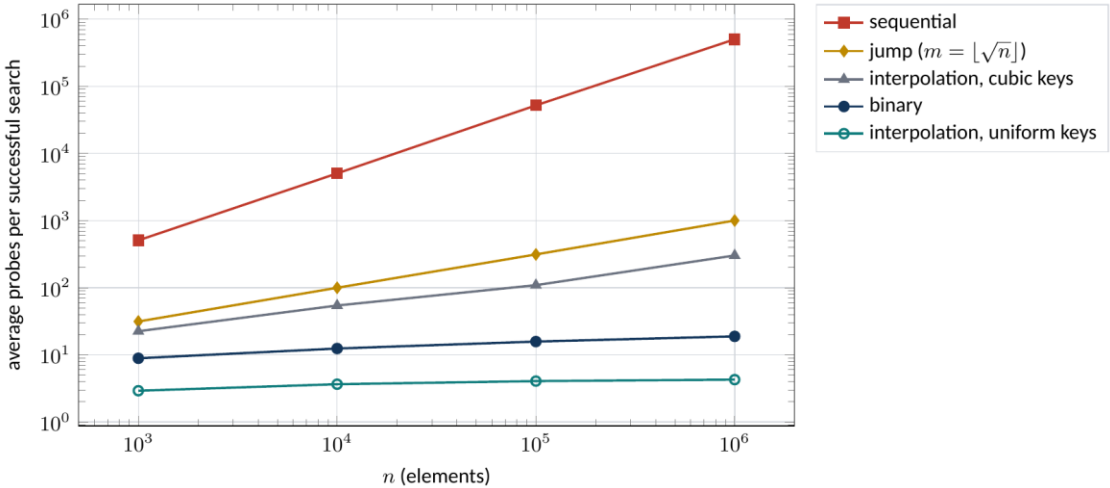


Figure 2.11 Average probes per successful search as n grows from 10^3 to 10^6, on logarithmic axes. Sequential search rises with slope 1, jump search with slope 1/2; binary search rises by about 3.3 probes for every tenfold increase; interpolation search on uniform keys barely rises at all, but on cubic keys it climbs steeply.

***Figure 2.11**: sequential rises with slope 1, jump with slope 1/2, binary by about 3.3 per tenfold step; interpolation on uniform keys barely rises, but on cubic keys it climbs from 22 to 301. Book: Section 2.10.4.*

Searching With the Standard Libraries

Table 2.4 Searching with the standard libraries of C++, Java and Python.

Question	C++ (`<algorithm>`)	Java (`java.util`)	Python
Linear search	`std::find` → iterator, `end()` if absent	`list.indexOf(x)` → index or −1	`x in list`; `list.index(x)` raises `ValueError`
Present in sorted data?	`std::binary_search` → `bool`	`Arrays.binarySearch` → ≥ 0 if found	`i = bisect_left(a, x)`, then `a[i] == x`
First position ≥ x	`std::lower_bound`	`-(r) - 1` when $r < 0$ (see text)	`bisect.bisect_left`
First position > x	`std::upper_bound`	—	`bisect.bisect_right`
All copies of x	`std::equal_range`	—	`bisect_left` … `bisect_right`
Insert, keeping order	`v.insert(upper_bound(…), x)`	`list.add(-(r) - 1, x)`	`bisect.insort(a, x)`
Search records by a field	comparator lambda	`Collections.binarySearch(list, key, comparator)`	`key=` argument (Python 3.10+)

***Table 2.4**: in production code, call the library — but know exactly what each call promises. C++ works on half-open ranges and guarantees about $\log_2 n$ comparisons; Python's `list.index` raises `ValueError` instead of returning −1. Book: Section 2.11.*

Java: One `int`, Two Answers

Listing 2.5 From `LibrarySearch.java`: `Arrays.binarySearch` and its negative result, and inserting at the insertion point.

```
        int r = Arrays.binarySearch(codes, 60);
        System.out.println("Arrays.binarySearch(codes, 60) = " + r +
                ", so 60 would be inserted at " + (-r - 1));

        int[] years = {1998, 2001, 2001, 2005, 2005, 2005, 2009, 2012, 2012, 2020};
        System.out.println("Arrays.binarySearch(years, 2005) = " +
                Arrays.binarySearch(years, 2005) +
                " (one of the 2005s, no promise which)");
        List<Integer> list = new ArrayList<>();
        for (int y : years) list.add(y);
        int pos = Collections.binarySearch(list, 2003);
        // keeps the list sorted
        list.add(-pos - 1, 2003);
        System.out.println("after inserting 2003: " + list);
```

```
Arrays.binarySearch(codes, 88) = 6
Arrays.binarySearch(codes, 60) = -5, so 60 would be inserted at 4
Arrays.binarySearch(years, 2005) = 4 (one of the 2005s, no promise which)
after inserting 2003: [1998, 2001, 2001, 2003, 2005, 2005, 2005, 2009, 2012, 2012, 2020]
```

Listing 2.5*: a negative result r means "absent, and would go at $-r-1$"; the extra -1 keeps an absent key from looking like "found at index 0". Java has no upper bound: write* ***Pseudocode 2.5*** *with* `<=`*. Book: Section 2.11.2.*

Python: the `bisect` Module

Listing 2.6 From `library_search.py`: lower and upper bound by `bisect_left` and `bisect_right`, `insort`, and a search of records with `key=`.

```
    years = [1998, 2001, 2001, 2005, 2005, 2005, 2009, 2012, 2012, 2020]
    lo, hi = bisect.bisect_left(years, 2005), bisect.bisect_right(years, 2005)
    print(
        "2005: bisect_left = %d, bisect_right = %d, count = %d" % (lo, hi, hi - lo)
    )
    bisect.insort(years, 2003)  # keeps the list sorted
    print("after insort(2003):", years)
    i = bisect.bisect_left(shelf, 63, key=lambda book: book[0])  # Python 3.10+
    print("record with code 063:", shelf[i][1])
```

```
2005: bisect_left = 3, bisect_right = 6, count = 3
after insort(2003): [1998, 2001, 2001, 2003, 2005, 2005, 2005, 2009, 2012, 2012, 2020]
record with code 063: Advanced JSP & Servlet
```

***Listing 2.6**: `insort` finds the place in $O(\lg n)$ but the insertion still moves every later element, $O(n)$. The `key=` argument (Python 3.10+) is applied to the elements, so we pass the bare code 63. Book: Section 2.11.3.*

Choosing a Method

Table 2.5 The search methods of this chapter, what each needs, and what each costs.

Method	Needs	Best	Average	Worst	Extra space
Sequential	nothing	$O(1)$	$(n+1)/2$ probes	n probes	$O(1)$
Sentinel	one writable spare cell	$O(1)$	$(n+1)/2$ probes	$n+1$ probes	$O(1)$
Binary	sorted data	$O(1)$	$\approx \lg n - 1$ probes	$\lfloor \lg n \rfloor + 1$ probes	$O(1)$ iterative
Lower / upper bound	sorted data	$\approx \lg n$	$\approx \lg n$	$\lceil \lg(n+1) \rceil$ probes	$O(1)$
Interpolation	sorted numeric, evenly spread keys	$O(1)$	$\approx \lg\lg n$ (uniform)	$O(n)$	$O(1)$
Jump	sorted data	$O(1)$	$\approx \sqrt{n}$	$\approx 2\sqrt{n}$	$O(1)$
Staircase (2-D)	rows and columns sorted	$O(1)$	—	$R + C - 1$ probes	$O(1)$

***Table 2.5**: unsorted and searched once, sequential; searched often, sort once or hash; sorted, binary search, in its lower-bound form for duplicates and ranges; interpolation only on measured, evenly spread numbers; jump only where backward moves are expensive. Book: Section 2.12.1.*

Nothing That Compares Beats Binary Search

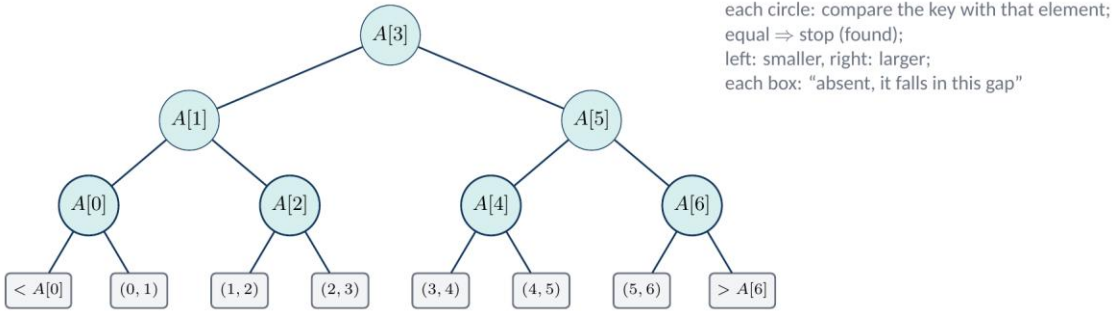


Figure 2.12 The decision tree of binary search on seven sorted elements. Each circle compares the key with one element; each box is one of the eight gaps into which an absent key can fall. Every root-to-box path has three comparisons.

$$h \geq \lceil \lg(n+1) \rceil. \tag{2.11}$$

Figure 2.12*: an absent key can fall into $n+1$ gaps, so the decision tree needs $n+1$ leaves and $2^h \geq n+1$, which is Equation (2.11). Since $\lceil lg(n+1) \rceil = \lfloor lgn \rfloor + 1$, binary search is optimal. To beat it you need arithmetic on keys, or hashing (Chapter 15). Book: Section 2.12.2.*

Profitina in Practice

Time series are sorted, so "scans between 1 July and 30 September" is two lower-bound searches, $O(\lg n)$, however long the history.

"N new since you last looked" is $n - upper(t)$; **a rank on a leaderboard** is an insertion point.

Screening links asks "have we seen this before?" many times a second: $\lfloor \lg n \rfloor + 1$ probes, not n.

Keep data in an order that matches the questions you will ask, and every question becomes a bound search.

Profitina's implementation details are its own; the shape of these problems is universal. Book: Section 2.13.1.

Case Study: Profitina Laundry

Listing 2.7 From `laundry_orders.cpp`: the orders between 10:00 and 12:00, found by two lower bounds.

```
    int first = lowerBound(times, minutes("10:00")),
        stop = lowerBound(times, minutes("12:00"));
    double total = 0;
    for (int i = first; i < stop; i++) total += ORDERS[i].kg;
    printf("orders from 10:00 to before 12:00: %d orders, %.1f kg\n", stop - first,
           total);
    for (int i = first; i < stop; i++)
        printf("  #%d %s %-5s %.1f kg\n", ORDERS[i].number, ORDERS[i].time.c_str(),
               ORDERS[i].customer.c_str(), ORDERS[i].kg);
```

```
orders from 10:00 to before 12:00: 4 orders, 21.0 kg
  #1005 10:05 Rina  4.0 kg
  #1006 10:40 Joko  6.0 kg
  #1007 11:15 Maya  3.0 kg
  #1008 11:50 Tono  8.0 kg
```

***Listing 2.7**: orders arrive in time order with consecutive numbers, so the list is sorted for free. Times become minutes since midnight before searching, since `"9:30" > "10:05"` as text. The search by customer name is a scan, because names are not sorted. Book: Section 2.13.2.*

Common Mistakes

Binary search on unsorted data, or **mixing interval conventions**, or `low = mid` on a closed interval.

`(low + high) / 2`; expecting the *first* copy from a plain binary search; misreading Java's negative result.

Returning a flag instead of a position; stopping early when counting; a sentinel on shared or read-only data.

Interpolation without its guards, or trusted on unmeasured data (17 times worse on the real catalog).

Adding a count to an address; scanning a grid column by column; comparing times as text; believing probes are times.

Fifteen mistakes, each with its fix, are listed in the book. Chapter 4 works problems of exactly this kind. Book: Section 2.14.

Summary: Five Things to Remember

1. Return a position or -1; count probes. Sequential search costs $(n+1)/2$ on average and n when absent: $O(n)$.

2. Binary search keeps the invariant of **Figure 2.4** and needs at most $\lfloor \lg n \rfloor + 1$ probes (Equation (2.5)).

3. One interval convention, a shrinking update, and the safe middle of Equation (2.6); reach for lower and upper bound first.

4. Interpolation search is $\lg\lg n$ on even keys and $O(n)$ on bunched ones; jump search needs about $2\sqrt{n}$.

5. No comparison search beats $\lceil \lg(n+1) \rceil$ (Equation (2.11)); measure, because probes predict growth, not speed.

Binary search, interpolation and jump search all assumed that the data had already been sorted. Book: Section 2.15.

Next Time

Chapter 3 — Sorting. The classic sorts, from bubble sort to quicksort, merge sort and heapsort, measured on the same catalog.

Sorting once so that you can binary-search many times is one of the best bargains in computing.

Before then: work Exercises 2.4, 2.6, 2.7 and 2.11, in C++ and in one other language.

Exercise 2.6 hides several bugs in one short binary search: find each with the four traps of today. Book: Sections 2.16 and 2.17.

Lecture 3 — Sorting

Course: Data Structure (EF234201)

Book: Irfan Subakti, *Data Structure with C++, Java and Python*, Chapter 3

Lecturer: Irfan Subakti — yifana@gmail.com

Today we pay the debt that binary search left: eleven ways to put data in order, from the sort anybody invents to the ones inside the libraries we use every day. Book: Chapter 3, Sections 3.1–3.17.

What This Meeting Covers

1. The words we need — sorted order, the swap, comparisons and swaps, inversions; in-place, stable, adaptive.

2. The simple sorts — bubble (with early exit), exchange, selection and insertion sort; why all four are quadratic.

3. Faster by long jumps — Shell sort and its gap sequences.

4. Divide and conquer — merge sort and quicksort with Lomuto's and Hoare's partitions; heap sort, a first look.

5. Past the $n \lg n$ barrier — counting sort and radix sort.

6. The complete table, the measurements, the library sorts of 2026, Profitina Laundry.

Every algorithm sorts the same six numbers, so the traces can be laid side by side. Book: Section 3.1.

Where We Left Off

Binary search finds one book among a million in about twenty looks — but only on data that is **already sorted**. On an unsorted shelf it quietly returns wrong answers.

Each of today's sorts teaches an idea that returns later: the adjacent swap, adaptivity, divide and conquer (Chapter 10), the heap (Chapter 14).

The running array, positions numbered from 0:

$$A = [22, 10, 15, 3, 8, 2]$$

Chapter 2 sorted its arrays with the library and promised to explain how sorting is done. Book: Section 3.1.

What Sorting Is

DEFINITION. Given n items $a_0, \ldots, a_{n-1}$ and an order on their keys, **sorting** produces a rearrangement $a'_0, \ldots, a'_{n-1}$ of the same items such that Equation (3.1) holds. Descending order uses $\geq$.

$$a'_0 \leq a'_1 \leq \cdots \leq a'_{n-1} \tag{3.1}$$

Two conditions: *in order*, and *a rearrangement* — six copies of the smallest number are in order but are not a sort.

The **key** is the part of the item the order looks at: title today, category tomorrow.

Sorted data makes searching, finding duplicates, grouping, merging and ranking cheap.

Knuth devotes most of a 780-page volume to sorting; the ideas are clean, well understood and well documented. Book: Sections 3.2.1 and 3.2.2.

Six Families of Sorting Methods

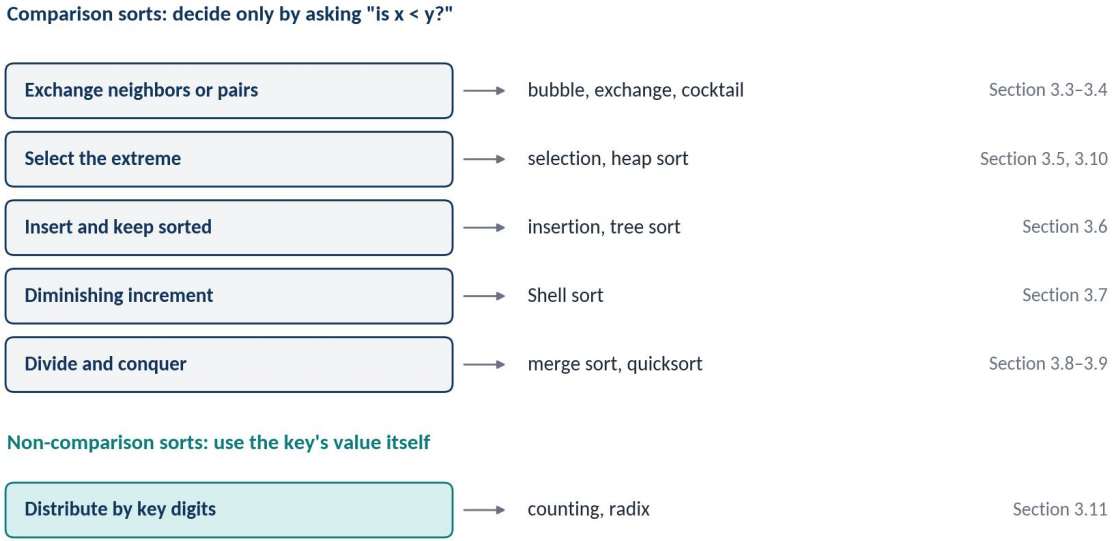


Figure 3.1 The families of sorting methods in this chapter. The first five compare keys; the last uses the digits of the keys directly.

Figure 3.1: exchange, selection, insertion, diminishing increment, divide and conquer, and distribution. The first five compare keys; the last uses the digits of the keys directly. Book: Section 3.2.3.

The One Move Every Sort Needs

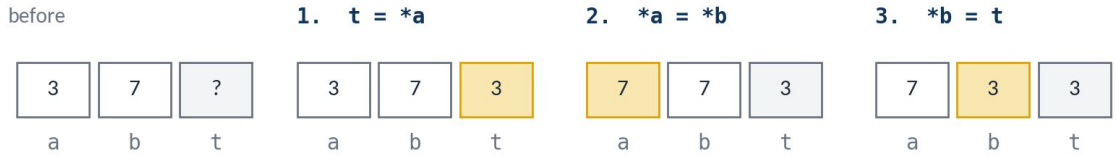


Figure 3.2 Exchanging two values with a spare variable. Each highlighted box is the one that changes in that step.

Figure 3.2: `a = b; b = a;` *destroys the only copy of the first value. A temporary* `t` *keeps it safe: three assignments. Book: Section 3.2.4.*

Swapping Through Pointers

Listing 3.1: a function that receives two `int`s receives copies; to exchange the caller's variables it receives their **addresses**.

Listing 3.1 From `swap_demo.cpp`: exchanging two `int`s through pointers.

```
void swapInts(int* a, int* b) {        // receives two addresses, not two values
    int t = *a;                         // keep the first value safe
    *a = *b;
    *b = t;
}
int main() {
    int a = 3, b = 7;
    std::printf("before: a = %d, b = %d\n", a, b);
    swapInts(&a, &b);                   // pass the addresses of a and b
    std::printf("after:  a = %d, b = %d\n", a, b);
```

Java cannot exchange two local variables in a method, but can exchange two slots of an array it receives. **Python** writes `a, b = b, a`.

In real C++ we call `std::swap(x, y)`, which takes references and does the same three assignments; from bubble sort on, our C++ programs use it. Book: Section 3.2.4.

Counting the Work

A **comparison** asks “is x before y?”; a **swap** exchanges two items; a **shift** or **move** copies one. A swap is about three moves.

Table 3.1 How $n \lg n$ and $n^2/2$ grow. At a billion items, a computer doing 10^9 steps per second needs 30 seconds for $n \lg n$ steps and 16 years for $n^2/2$ steps.

n	$n \lg n$	$n^2/2$	Ratio
10	33	50	1.5
$1,000$	$9,966$	$500,000$	50
$1,000,000$	$19,931,569$	5×10^{11}	$25,000$
10^9	2.99×10^{10}	5×10^{17}	1.7×10^7

Table 3.1: at a billion items and 10^9 steps a second, $n \lg n$ takes 30 seconds and $n^2/2$ takes 16 years. Doubling n quadruples a quadratic sort and slightly more than doubles an $n \lg n$ one. Book: Section 3.2.5.

Inversions: How Unsorted Is It?

DEFINITION. An **inversion** is a pair (i, j) with $i < j$ and $a_i > a_j$; Equation (3.2) counts them.

$$inv(a) = \left|\left\{(i, j) : 0 \le i < j \le n-1, a_i > a_j\right\}\right|. \tag{3.2}$$

Sorted: 0 inversions. Reversed: all $n(n-1)/2$ pairs.

Worked count on A: 22 beats five later numbers, 10 beats three (3, 8, 2), 15 beats three, 3 beats one, 8 beats one:

$$5+3+3+1+1+0=13$$

Keep 13 in mind: it reappears as the swaps of bubble sort and the shifts of insertion sort. Book: Section 3.2.5.

Three Properties

> **DEFINITION.** A sort is **in-place** when it uses only a constant amount of memory besides the array. It is **stable** when items with equal keys come out in their input order. It is **adaptive** when it does less work on nearly sorted input.

In-place matters when memory is short or the data huge.

Stable matters whenever a record is sorted by one key after being arranged by another: the catalog by id, then by category.

Adaptive matters because real data is often *almost* sorted: yesterday's list plus a few new entries.

Section 3.12 shows an unstable sort scrambling the catalog. Book: Section 3.2.6.

Bubble Sort

Walk left to right, swapping neighbors that are out of order. One **pass** carries the largest element to the end, like a bubble rising through soda.

Invariant: after pass p, the last p positions hold the p largest elements, in order.

Early exit: a pass without a swap means the array is sorted, by Equation (3.1).

Pseudocode 3.1 Bubble sort with early exit: each pass carries the largest remaining element to the end, and a pass without a swap ends the sort.

```
BUBBLE-SORT(A, n)
 1  for pass = 1 to n − 1
 2      swapped = FALSE
 3      for j = 0 to n − pass − 1          // the tail is already in place
 4          if A[j] > A[j+1]
 5              exchange A[j] with A[j+1]
 6              swapped = TRUE
 7      if swapped == FALSE                  // a pass with no swap: sorted
 8          return
```

Pseudocode 3.1: the flag `swapped` is reset at the start of every pass. Book: Sections 3.3.1 and 3.3.2.

Bubble Sort, Traced

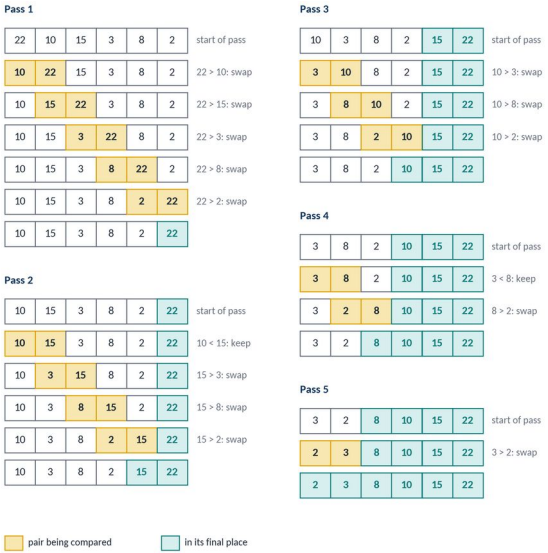


Figure 3.3 Bubble sort on $A = [22, 10, 15, 3, 8, 2]$, one row per comparison. Pass 1 carries 22 to the end, pass 2 carries 15, and so on; five passes and fifteen comparisons in all.

Figure 3.3: pass 1 carries 22 to the end, pass 2 carries 15; five passes, fifteen comparisons. Small numbers crawl left one step per pass. Book: Section 3.3.3.

Bubble Sort in C++

Listing 3.2 From `bubble_sort.cpp`: the two loops, the direction flag and the early exit.

```
    for (int pass = 1; pass < n; ++pass) {
        bool swapped = false;
        for (int j = 0; j < n - pass; ++j) {  // the last pass-1 slots are final
            ++comparisons;
            bool outOfOrder = ascending ? a[j] > a[j + 1] : a[j] < a[j + 1];
            if (outOfOrder) {
                swap(a[j], a[j + 1]);
                ++swaps;
                swapped = true;
            }
        }
        if (!swapped) break;  // early exit: nothing moved
```

```
pass 1: [10 15 3 8 2 22]
pass 2: [10 3 8 2 15 22]
pass 3: [3 8 2 10 15 22]
pass 4: [3 2 8 10 15 22]
pass 5: [2 3 8 10 15 22]
result [2 3 8 10 15 22], comparisons = 15, swaps = 13
```

Listing 3.2 prints the array after each pass, matching Figure 3.3. On sorted input it stops after one pass of 5 comparisons; on reversed input every comparison is a swap. Java writes the swap out with `t`; Python calls the counter `pass_no`, since `pass` is a keyword. Book: Section 3.3.4.

How Much Work, Exactly

Worst case, Equation (3.3); for $n = 6, 15$; for $n = 100,000$, about five billion:

$$C_{\text{worst}}(n) = \sum_{p=1}^{n-1} (n-p) = (n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2}. \tag{3.3}$$

Best case with the early exit: $n-1$ comparisons. **Swaps**: an adjacent swap removes exactly one inversion, Equation (3.4):

$$\text{swaps made by bubble sort} = inv(A). \tag{3.4}$$

Average: each pair is inverted in half of all orders, Equation (3.5):

$$\overline{inv}(n) = \frac{1}{2} \cdot \frac{n(n-1)}{2} = \frac{n(n-1)}{4}. \tag{3.5}$$

> **KEY IDEA.** Any algorithm that swaps only *adjacent* elements needs $n(n-1)/4$ swaps on average: quadratic. To beat n^2, move elements over long distances.

Bubble sort is in-place and stable: it swaps only on a strict `>`*. Book: Sections 3.3.5 and 3.3.6.*

Exchange Sort

Pseudocode 3.2 Exchange sort: position i is compared with every later position and takes any smaller value it meets.

```
EXCHANGE-SORT(A, n)
 1  for i = 0 to n − 2
 2      for j = i + 1 to n − 1
 3          if A[j] < A[i]
 4              exchange A[i] with A[j]
```

Position i is compared with **every** later position and takes any smaller value it meets.

$n(n-1)/2$ comparisons in *every* case: no early exit, not adaptive.

Its 13 swaps on A are a coincidence, not a law: a long-distance swap can change many inversions at once.

*Pseudocode 3.2 is in-place but **not stable**: a long jump can carry an element past an equal key. Book: Section 3.4.*

Exchange Sort, Traced

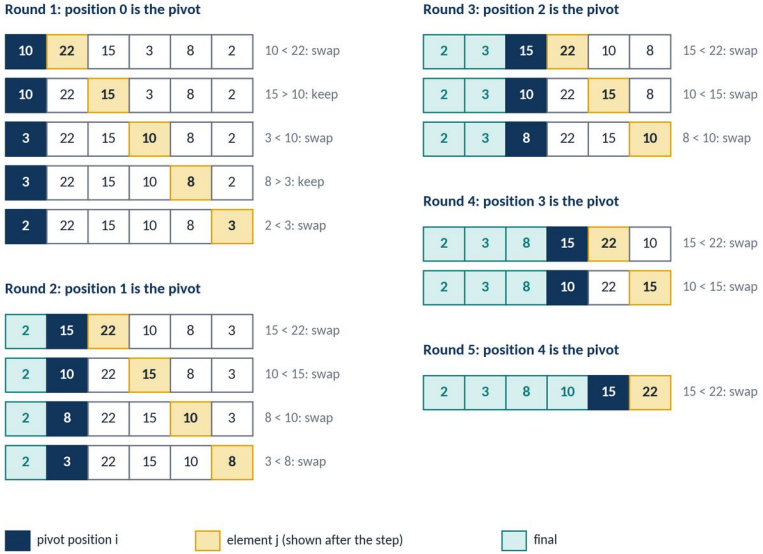


Figure 3.4 Exchange sort on A. In round 1 the value at position 0 changes three times, $22 \to 10 \to 3 \to 2$, before the smallest value settles there.

Figure 3.4: in round 1 position 0 changes three times, $22 \to 10 \to 3 \to 2$, before the smallest value settles there. Book: Section 3.4.

Selection Sort: Search First, Swap Once

Pseudocode 3.3 Selection sort: find the smallest remaining element, then move it into place with one exchange.

```
SELECTION-SORT(A, n)
 1  for i = 0 to n − 2
 2      pos = i                                  // smallest seen so far
 3      for j = i + 1 to n − 1
 4          if A[j] < A[pos]
 5              pos = j                          // remember it; move nothing yet
 6      if pos ≠ i
 7          exchange A[i] with A[pos]
```

```
  round 1: smallest is 2 (was at index 5) -> [2 10 15 3 8 22]
  round 2: smallest is 3 (was at index 3) -> [2 3 15 10 8 22]
  round 3: smallest is 8 (was at index 4) -> [2 3 8 10 15 22]
  round 4: smallest is 10 (was at index 3) -> [2 3 8 10 15 22]
  round 5: smallest is 15 (was at index 4) -> [2 3 8 10 15 22]
  comparisons = 15, swaps = 3
```

Pseudocode 3.3 is sorting built from searching: each round is the linear scan of Chapter 2. Always $n(n-1)/2$ *comparisons, but* ***at most*** $n-1$ ***swaps*** *— 3 on* A *against 13. Good when moving is expensive (large records, flash memory); not adaptive, not stable. Book: Sections 3.5.1–3.5.3.*

Selection Sort, Traced

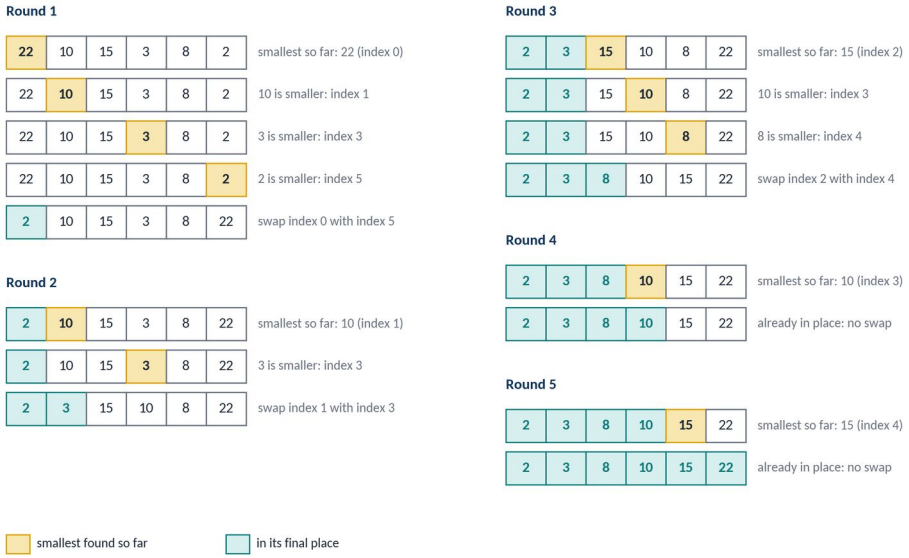


Figure 3.5 Selection sort on A. Round 1 finds the minimum 2 at index 5 and makes one exchange; rounds 4 and 5 find the minimum already in place and make none.

Figure 3.5: round 1 finds the minimum 2 at index 5 and makes one exchange; rounds 4 and 5 find the minimum already in place and make none. Book: Section 3.5.1.

Insertion Sort: a Hand of Cards

Pseudocode 3.4 Insertion sort: grow a sorted prefix by inserting one key at a time, shifting larger elements right.

```
INSERTION-SORT(A, n)
 1  for i = 1 to n − 1
 2      key = A[i]                          // the card in our hand
 3      j = i − 1
 4      while j ≥ 0 and A[j] > key          // test j ≥ 0 first
 5          A[j+1] = A[j]                   // shift the larger element right
 6          j = j − 1
 7      A[j+1] = key                        // drop the key into the hole
```

Invariant: at the start of step i, $A[0..i-1]$ holds its original elements, sorted.

> **TEST THE INDEX FIRST.** `while (a[j] > key && j >= 0)` reads `a[-1]`: undefined behavior in C++, an exception in Java, and in Python silently the *last* element. Write `j >= 0 && a[j] > key`.

Pseudocode 3.4 relies on short-circuit evaluation: `&&` and `and` skip the right operand when the left is false. Book: Sections 3.6.1 and 3.6.2.

Insertion Sort, Traced

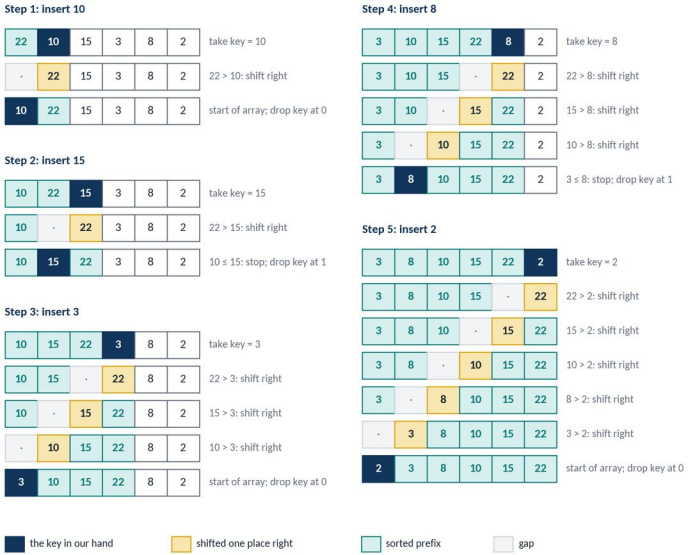


Figure 3.6 Insertion sort on A. Each step takes a key, shifts the larger elements of the sorted prefix one place right, and drops the key into the gap.

Figure 3.6: each step takes a key (navy), shifts the larger elements of the sorted prefix one place right, and drops the key into the gap. Book: Section 3.6.1.

Insertion Sort in Python

Listing 3.3 From `insertion_sort.py`: the loop tests `j >= 0` before reading `a[j]`, and stops at an equal key.

```python
def insertion_sort(a, trace=False):
    comparisons = shifts = 0
    for i in range(1, len(a)):
        key = a[i]                      # the card in our hand
        j = i - 1
        while j >= 0:                   # test j >= 0 BEFORE reading a[j]
            comparisons += 1
            if a[j] <= key:             # equal keys stay put: stable
                break
            a[j + 1] = a[j]             # shift the larger one right
            shifts += 1
            j -= 1
        a[j + 1] = key
```

```
Insertion sort on [22 10 15 3 8 2]
  insert 10 at index 0: [10 22 15 3 8 2]
  insert 15 at index 1: [10 15 22 3 8 2]
  insert 3 at index 0: [3 10 15 22 8 2]
  insert 8 at index 1: [3 8 10 15 22 2]
  insert 2 at index 0: [2 3 8 10 15 22]
```

Listing 3.3: 15 comparisons and 13 shifts on A — the 13 inversions again. The C++ and Java versions print the same lines. Book: Section 3.6.3.

Why Insertion Sort Is the Best Simple Sort

Each shift fixes exactly one inversion: shifts $= inv(A)$. Each step adds at most one comparison that stops the walk, Equation (3.6):

$$n-1 \le C_{\text{insertion}}(A) \le inv(A) + n - 1 \le \frac{n(n-1)}{2} + n - 1. \tag{3.6}$$

Sorted input: $n-1$ comparisons, no shifts. Reversed: $n(n-1)/2$ of each. Random: about $n^2/4$.

The work is proportional to $n + inv(A)$: that is the precise meaning of **adaptive**.

In-place, stable, tiny, and fast on small or nearly sorted arrays — which is why every serious library sort finishes by running insertion sort on small pieces. Book: Section 3.6.4.

Shell Sort: Long Jumps First

Pseudocode 3.5 Shell sort: insertion sort on elements h apart, for a decreasing sequence of gaps ending with 1.

```
SHELL-SORT(A, n, gaps)                      // gaps decreasing, last gap = 1
 1  for each gap h in gaps
 2      for i = h to n − 1                   // insertion sort with step h
 3          key = A[i]
 4          j = i − h
 5          while j ≥ 0 and A[j] > key
 6              A[j+h] = A[j]
 7              j = j − h
 8          A[j+h] = key
```

Pseudocode 3.5, Donald Shell, 1959: insertion sort on elements h apart for decreasing gaps. With gap 1 it is exactly insertion sort, so it is always correct; the earlier gaps only make the last pass cheap. Not stable. Book: Section 3.7.1.

Worked Example 1: Gaps 3 and 1

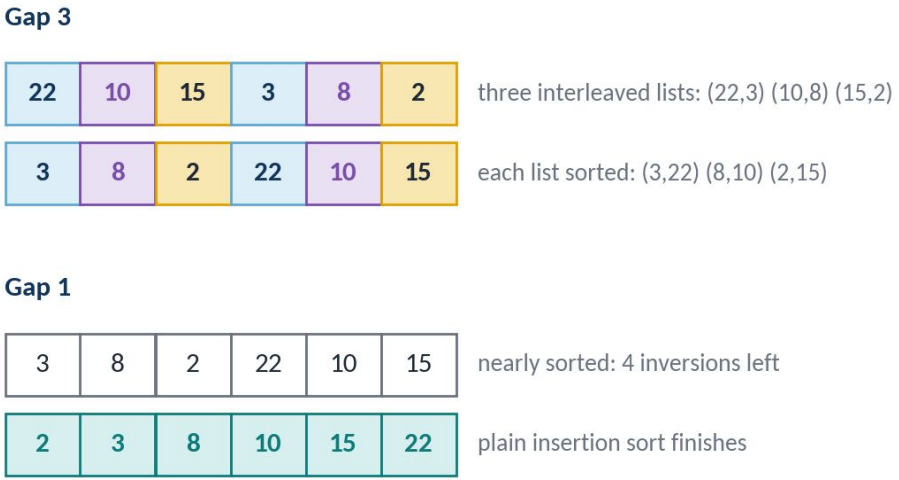


Figure 3.7 Shell sort on A with gaps 3 and 1. Sorting the three interleaved pairs removes 9 of the 13 inversions at once; insertion sort then removes the remaining 4.

Figure 3.7: gap 3 sorts the pairs $(22,3)$, $(10,8)$, $(15,2)$; *22 jumps three places in one move and the array becomes* $[3,8,2,22,10,15]$, *removing 9 of the 13 inversions. Gap 1 removes the other 4: 11 comparisons against 15. Book: Section 3.7.2.*

The Gap Sequence Is Everything

Shell's $\lfloor n/2 \rfloor, \lfloor n/4 \rfloor, \ldots, 1$: worst case still $\Theta(n^2)$.

Knuth's $1, 4, 13, 40, 121, \ldots$, Equation (3.7), worst case $O(n^{3/2})$:

$$h_1 = 1, h_{k+1} = 3h_k + 1, \tag{3.7}$$

Ciura's $1, 4, 10, 23, 57, 132, 301, 701$ (2001, found by experiment): the best known in practice.

```
Comparisons on 100000 random numbers:
  Shell n/2: 16 gaps, 4488726 comparisons
  Knuth 3h+1: 11 gaps, 3842607 comparisons
  Ciura: 13 gaps, 2554426 comparisons
```

Insertion sort would need about 2.5 billion comparisons on these 100,000 numbers; Shell sort needs 2.6 to 4.5 million — a thousand times less work from a change of eight lines. Book: Section 3.7.3.

Merge Sort: Divide and Conquer

Pseudocode 3.6 `MERGE` and `MERGE-SORT`. The buffer T has the same length as A and is allocated once, before the first call.

```
MERGE(A, T, lo, mid, hi)                        // both runs are sorted
 1  i = lo; j = mid; k = lo
 2  while i < mid and j < hi
 3      if A[i] ≤ A[j]                           // ≤ keeps ties stable
 4          T[k] = A[i]; i = i + 1
 5      else
 6          T[k] = A[j]; j = j + 1
 7      k = k + 1
 8  copy the rest of A[i..mid−1], then of A[j..hi−1], to T[k..hi−1]
 9  copy T[lo..hi−1] back to A[lo..hi−1]

MERGE-SORT(A, T, lo, hi)                        // sorts A[lo..hi−1]
 1  if hi − lo < 2
 2      return                                   // 0 or 1 element: sorted
 3  mid = lo + ⌊(hi − lo)/2⌋
 4  MERGE-SORT(A, T, lo, mid)
 5  MERGE-SORT(A, T, mid, hi)
 6  MERGE(A, T, lo, mid, hi)
```

Pseudocode 3.6: ***divide*** *in half,* ***conquer*** *each half recursively,* ***combine*** *by merging — take the smaller top card of two sorted piles, repeat. One buffer T, allocated once;* `lo + (hi − lo)/2` *cannot overflow. Book: Section 3.8.1.*

Merge Sort, Traced

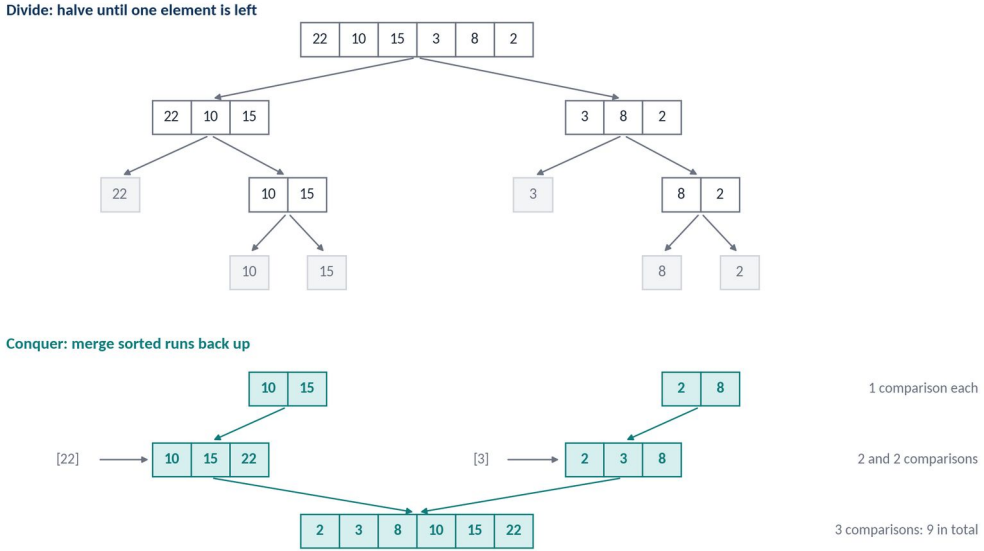


Figure 3.8 Merge sort on A. Dividing (top) splits the array until every piece has one element; merging (bottom) combines sorted runs, and the comparisons at each level are noted on the right.

Figure 3.8: the upper half divides, the lower half merges; $1+2+1+2+3=9$ *comparisons. The last merge takes 2, 3 and 8, and then copies 10, 15, 22 without comparing. Book: Section 3.8.2.*

`merge` in C++

Listing 3.4 From `merge_sort.cpp`: merging through one shared buffer, and the recursion.

```
// Merges the sorted runs a[lo..mid-1] and a[mid..hi-1] through the buffer tmp.
void merge(vector<int>& a, vector<int>& tmp, int lo, int mid, int hi) {
    int i = lo, j = mid, k = lo;
    while (i < mid && j < hi) {
        ++comparisons;
        // <= takes the left one on ties: stable
        if (a[i] <= a[j]) tmp[k++] = a[i++];
        else              tmp[k++] = a[j++];
    }
    while (i < mid) tmp[k++] = a[i++];
    while (j < hi)  tmp[k++] = a[j++];
    for (k = lo; k < hi; ++k) { a[k] = tmp[k]; ++moves; }
}

void mergeSort(vector<int>& a, vector<int>& tmp, int lo, int hi, int depth) {
    if (hi - lo < 2) return;  // 0 or 1 element: already sorted
    int mid = lo + (hi - lo) / 2;
    mergeSort(a, tmp, lo, mid, depth + 1);
    mergeSort(a, tmp, mid, hi, depth + 1);
```

Listing 3.4: `<=` *takes from the left run on ties, which makes merge sort* ***stable****. The program prints 9 comparisons and 16 moves for* A*. Book: Section 3.8.3.*

Why Merge Sort Is $n \lg n$, Always

Two half-size calls and a linear merge give Equation (3.8):

$$T(n) = 2T(n/2) + cn, T(1) = c. \tag{3.8}$$

Every level of the recursion tree costs cn, and there are $\lg n + 1$ levels, so Equation (3.9):

$$T(n) = cn(\lg n + 1) = \Theta(n \lg n). \tag{3.9}$$

Nothing depends on the data: $\Theta(n \lg n)$ in the best, average *and* worst case.

*The price is memory: a buffer of n elements, so merge sort is **not in-place**. Timsort and Powersort are merge sorts made adaptive by finding runs already in the data. Book: Section 3.8.4.*

Quicksort: Partition, Then Recurse

Pseudocode 3.7 `QUICKSORT` with a partition that returns the pivot's final index.

```
QUICKSORT(A, lo, hi)                          // sorts A[lo..hi], inclusive
 1  if lo < hi
 2      p = PARTITION(A, lo, hi)              // pivot now at index p
 3      QUICKSORT(A, lo, p − 1)
 4      QUICKSORT(A, p + 1, hi)
```

Pseudocode 3.8 Lomuto's partition: one left-to-right scan that grows a region of elements no larger than the pivot.

```
LOMUTO-PARTITION(A, lo, hi)
 1  x = A[hi]                                 // the pivot
 2  i = lo − 1                                // A[lo..i] ≤ x
 3  for j = lo to hi − 1
 4      if A[j] ≤ x
 5          i = i + 1
 6          exchange A[i] with A[j]
 7  exchange A[i+1] with A[hi]                // pivot between the regions
 8  return i + 1
```

Pseudocode 3.7, Hoare 1959: real work to divide, nothing to combine. Pseudocode 3.8 keeps $A[lo..i] \le x$ *and* $A[i+1..j-1] > x$*, then drops the pivot between the regions. Book: Sections 3.9.1 and 3.9.2.*

Lomuto's Partition, Traced



Figure 3.9 Lomuto's partition. Top: partitioning $[10, 15, 3, 8]$ around the pivot 8. Bottom: the four partitions that sort A with the last element as pivot.

Figure 3.9: with the last element as pivot, 2 and then 22 split nothing off one side; only the third pivot, 8, splits evenly. Book: Section 3.9.2.

Hoare's Partition

Pseudocode 3.9 Hoare's partition: two indices move toward each other and exchange pairs that are on the wrong sides.

```
HOARE-PARTITION(A, lo, hi)
 1  x = A[lo + ⌊(hi − lo)/2⌋]                  // the pivot value
 2  i = lo − 1; j = hi + 1
 3  loop
 4      repeat i = i + 1 until A[i] ≥ x
 5      repeat j = j − 1 until A[j] ≤ x
 6      if i ≥ j
 7          return j                            // A[lo..j] ≤ x ≤ A[j+1..hi]
 8      exchange A[i] with A[j]
```

i stops at an element $\geq x$, j at an element $\leq x$; both are on the wrong side, so they are exchanged.

The pivot need not end at j: the recursive calls are `QUICKSORT(A, lo, j)` and `QUICKSORT(A, j+1, hi)`.

Pseudocode 3.9 does about a third as many exchanges as Lomuto's and splits an array of equal keys evenly; library quicksorts descend from it. Book: Section 3.9.3.

Worked Example 2: Hoare on A

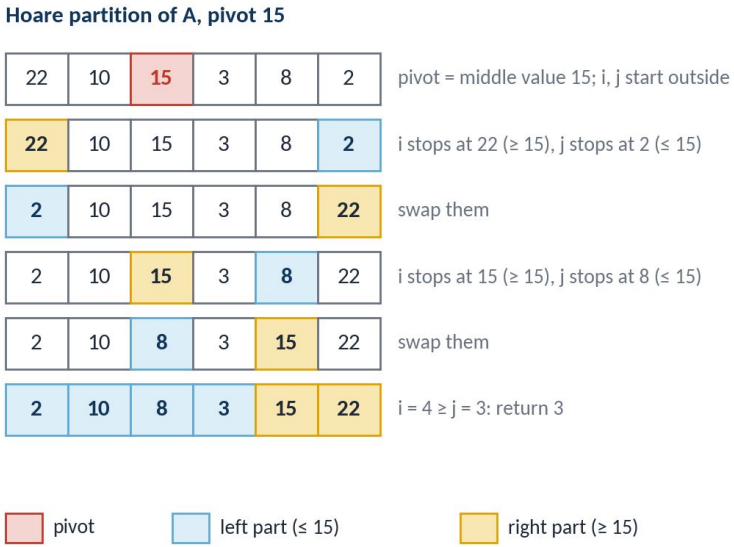


Figure 3.10 Hoare's partition of A with pivot 15. Two exchanges suffice; the indices cross with $j=3$, splitting A into $[2,10,8,3]$ and $[15,22]$.

Figure 3.10: pivot 15. Exchange 22 and 2; exchange 15 and 8; then $i=4\geq j=3$: the indices have crossed and the partition returns 3, splitting A into $[2,10,8,3]$ and $[15,22]$. Book: Section 3.9.3.

Lomuto's Partition in C++

Listing 3.5 From `quick_sort.cpp`: Lomuto's partition, counting comparisons and swaps.

```cpp
// Lomuto: pivot = a[hi]; a[lo..i] holds the elements <= pivot seen so far.
int lomutoPartition(vector<int>& a, int lo, int hi) {
    int pivot = a[hi];
    int i = lo - 1;
    for (int j = lo; j < hi; ++j) {
        ++comparisons;
        if (a[j] <= pivot) {
            ++i;
            if (i != j) { swap(a[i], a[j]); ++swaps; }
        }
    }
    if (i + 1 != hi) { swap(a[i + 1], a[hi]); ++swaps; }
    return i + 1;                                   // the pivot's final index
}
```

```
  sorted input, median of three: 28917 comparisons, depth 10
  sorted input, random: 41022 comparisons, depth 23
  random input, last element: 38004 comparisons, depth 23
  random input, median of three: 33207 comparisons, depth 18
  random input, random: 40509 comparisons, depth 27
```

Listing 3.5 on 3,000 elements: the last element as pivot turns sorted input into the worst case, $3000 \cdot 2999/2 = 4,498,500$ *comparisons and a recursion 2,998 deep. Java runs this on a thread with a 64 MB stack; Python raises its recursion limit. Book: Section 3.9.4.*

Best, Worst and Average

Worst: the pivot is always the smallest or largest, Equation (3.10):

$$T_{\text{worst}}(n) = T_{\text{worst}}(n-1) + (n-1) = \frac{n(n-1)}{2} = \Theta(n^2). \tag{3.10}$$

Average, random input or random pivot, Equation (3.11) — only 39% above the best case:

$$\overline{C}_{\text{quick}}(n) = 2(n+1)H_n - 4n \approx 2n \ln n \approx 1.39\, n \lg n, \tag{3.11}$$

Median of three turns sorted input into the best case; a **random pivot** makes every input equally good.

Recurse into the smaller side and the stack stays below $\lg n$*; switch to heap sort when the recursion gets too deep, and you have* ***introsort****, inside* `std::sort`*. Quicksort is not stable. Book: Section 3.9.5.*

Heap Sort: a Tree Inside an Array

Read $A[0..n-1]$ as a binary tree with the index arithmetic of Equation (3.12); no pointers are stored.

$$left(i)=2i+1, right(i)=2i+2, parent(i)=\lfloor (i-1)/2 \rfloor. \tag{3.12}$$

DEFINITION. An array is a **max-heap** when $A[parent(i)] \geq A[i]$ for every $i \geq 1$. The largest element is then at the root, $A[0]$.

Sift down: exchange a node with its larger child until it is at least as large as both, at most $2\lfloor \lg n \rfloor$ comparisons. **Build a heap**: sift down from index $\lfloor n/2 \rfloor - 1$ back to 0, in $O(n)$.

Heap sort keeps selection sort's plan — take the largest, put it at the end — but makes the largest free to find. Chapter 14 builds the heap in full. Book: Section 3.10.1.

SIFT-DOWN and HEAP-SORT

Pseudocode 3.10 SIFT-DOWN and HEAP-SORT: build a max-heap, then repeatedly move the root to the end of the shrinking heap and repair the root.

```
SIFT-DOWN(A, i, size)                          // heap is A[0..size−1]
 1  loop
 2      largest = i; l = 2i + 1; r = 2i + 2
 3      if l < size and A[l] > A[largest]  then largest = l
 4      if r < size and A[r] > A[largest]  then largest = r
 5      if largest == i  then return
 6      exchange A[i] with A[largest]
 7      i = largest

HEAP-SORT(A, n)
 1  for i = ⌊n/2⌋ − 1 down to 0               // build a max-heap
 2      SIFT-DOWN(A, i, n)
 3  for end = n − 1 down to 1
 4      exchange A[0] with A[end]              // the maximum goes to its place
 5      SIFT-DOWN(A, 0, end)                   // repair the smaller heap
```

Pseudocode 3.10: $O(n \lg n)$ *in the worst case and in-place — a combination no other sort of the chapter has. Its jumps from* i *to* $2i+1$ *are unkind to the cache, and it is not stable. Book: Section 3.10.1.*

Heap Sort, Traced

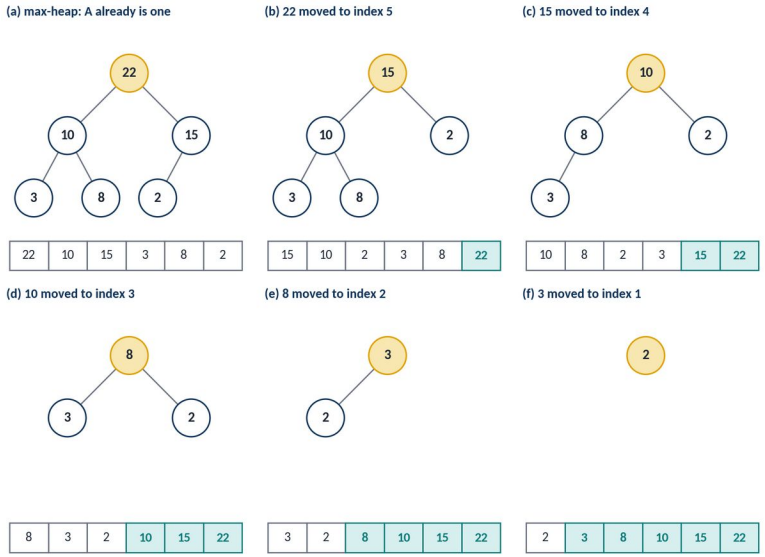


Figure 3.11 Heap sort on A. Each panel shows the heap as a tree (root in amber) and the whole array beneath it; the teal tail is the sorted part, which grows by one element per step.

Figure 3.11: A is already a max-heap. Step (b) exchanges 22 with 2 and sifts 2 down below 15, leaving the heap $[15,10,2,3,8]$; the teal tail grows by one element per step. Book: Section 3.10.2.

Sift Down in C++

Listing 3.6 From `heap_sort.cpp`: sift a node down below its larger child until the heap is repaired.

```cpp
// Lets a[i] sink until neither child is larger; the heap is a[0..size-1].
void siftDown(vector<int>& a, int i, int size) {
    while (true) {
        int largest = i, left = 2 * i + 1, right = 2 * i + 2;
        if (left < size) {
            ++comparisons;
            if (a[left] > a[largest]) largest = left;
        }
        if (right < size) {
            ++comparisons;
            if (a[right] > a[largest]) largest = right;
        }
        if (largest == i) return;
        swap(a[i], a[largest]);
        ++swaps;
        i = largest;
    }
}
```

Listing 3.6: the heap is `a[0..size-1]`, and the sorted tail lives in the same array behind it. On $[3,8,2,22,10,15]$ *the build phase moves 22 to the root first. Book: Section 3.10.2.*

The Barrier Comparison Sorts Cannot Cross

$n!$ possible orders must be told apart; k yes-or-no comparisons distinguish at most 2^k of them, hence Equation (3.13):

$$k \geq \lg(n!) \geq \lg\left(\left(\frac{n}{2}\right)^{n/2}\right) = \frac{n}{2}\lg\frac{n}{2} = \Omega(n \lg n). \tag{3.13}$$

For six elements, $\lg 720 \approx 9.49$: some input needs at least 10 comparisons, whatever the algorithm.

Merge sort and heap sort are optimal up to a constant factor.

The way past the barrier is to stop comparing, and to use the key values as array indices. Book: Section 3.11.1.

Counting Sort

Pseudocode 3.11 Counting sort for integer keys in $0..k-1$. No two keys are ever compared.

```
COUNTING-SORT(A, n, k)                          // keys are 0..k−1; output B
 1  count[0..k−1] = 0
 2  for i = 0 to n − 1
 3      count[A[i]] = count[A[i]] + 1
 4  for v = 1 to k − 1
 5      count[v] = count[v] + count[v−1]          // now: number of keys ≤ v
 6  for i = n − 1 down to 0                      // right to left keeps it stable
 7      count[A[i]] = count[A[i]] − 1
 8      B[count[A[i]]] = A[i]
 9  return B
```

$$T_{\text{counting}}(n,k)=\Theta(n+k). \tag{3.14}$$

$$T_{\text{radix}}(n)=\Theta(d(n+b)). \tag{3.15}$$

*Pseudocode 3.11: count, make running totals, place from **right to left** — stable. Equation (3.14) is linear when k is at most a constant times n; radix sort, Equation (3.15), is d passes of it in base b. Book: Section 3.11.2.*

Counting Sort, Traced

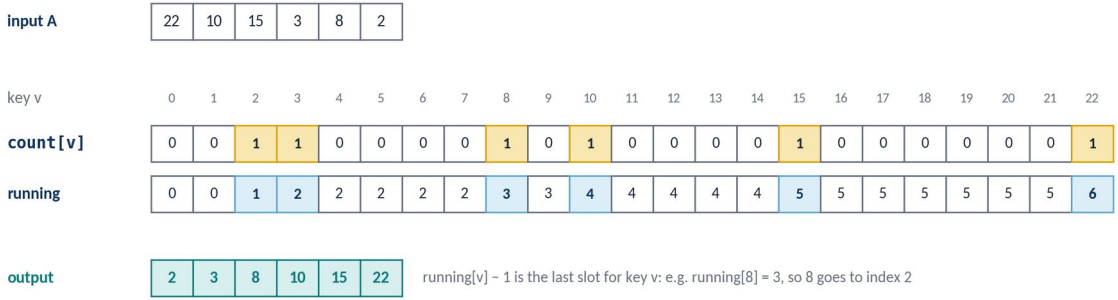


Figure 3.12 Counting sort on A with keys 0 to 22. The counts (amber) become running totals (blue), and each running total says where its key ends in the output.

Figure 3.12: the running total at 8 is 3 — three keys (2, 3, 8) are at most 8 — so 8 belongs at index $3-1=2$. *Book: Section 3.11.2.*

Radix Sort: Least Significant Digit First

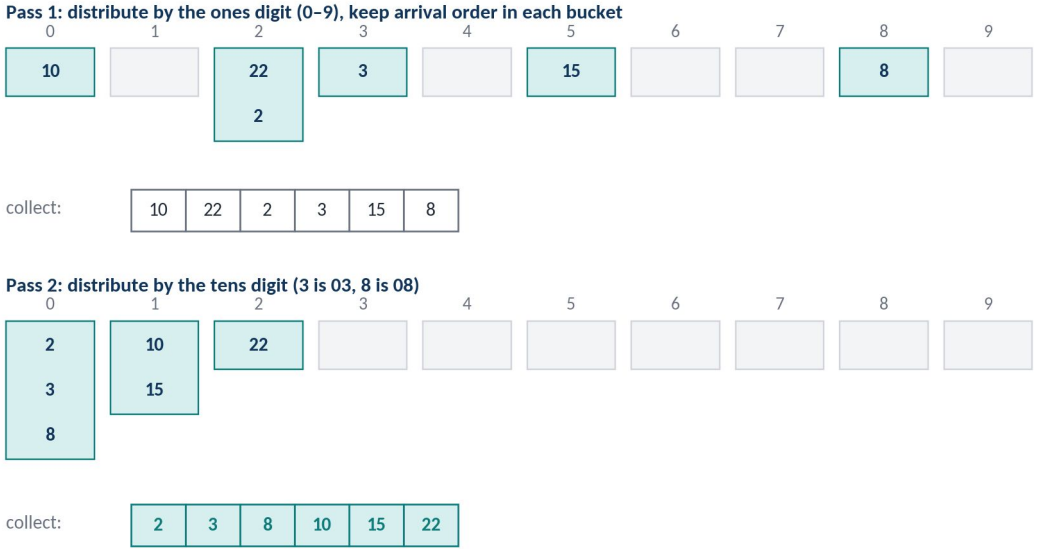


Figure 3.13 LSD radix sort on A. The first pass distributes by the ones digit; the second, by the tens digit, keeps each bucket in first-pass order, and the array comes out sorted.

Figure 3.13: sort by the ones digit, then stably by the tens digit; 22 stays ahead of 2 after the first pass, and 2, 3, 8 keep that order in the second. Book: Section 3.11.3.

Counting Sort in C++

Listing 3.7 From `counting_radix.cpp`: a stable counting sort parameterized by a key function, so one function serves radix sort too.

```cpp
// Stable counting sort of a by key(x) in 0..k-1; returns the sorted copy.
template <typename Key>
vector<int> countingSort(const vector<int>& a, int k, Key key, bool trace) {
    vector<int> count(k, 0);
    for (int x : a) ++count[key(x)];                // 1. how many of each key
    if (trace) cout << "  counts:        " << show(count) << "\n";
    for (int v = 1; v < k; ++v) count[v] += count[v - 1];  // 2. running totals
    if (trace) cout << "  running total: " << show(count) << "\n";
    vector<int> out(a.size());
    // 3. right to left: stable
    for (int i = static_cast<int>(a.size()) - 1; i >= 0; --i)
        out[--count[key(a[i])]] = a[i];
    return out;
}
```

```
Counting sort on [22 10 15 3 8 2] (keys 0..22)
  counts:        [0 0 1 1 0 0 0 0 1 0 1 0 0 0 0 1 0 0 0 0 0 0 1]
  running total: [0 0 1 2 2 2 2 2 3 3 4 4 4 4 4 5 5 5 5 5 5 5 6]
  result [2 3 8 10 15 22]
```

Listing 3.7: radix sort passes `[place](int x) { return x / place % 10; }` as the key. Both sorts need an output array (not in-place) and both are stable; they sort integers and things cut into digits, not arbitrary objects. Book: Section 3.11.3.

Stability You Can See

Input (id order)		Stable (insertion sort)		Not stable (selection sort)	
Fiksi	**B1001**	Anak-anak	B1006	Anak-anak	B1006
Fiksi	**B1002**	Fiksi	**B1001**	Fiksi	**B1002**
Fiksi	**B1003**	Fiksi	**B1002**	Fiksi	**B1003**
Sejarah	B1004	Fiksi	**B1003**	Fiksi	**B1001**
Teknologi	B1005	Fiksi	**B1008**	Fiksi	**B1008**
Anak-anak	B1006	Sains	B1007	Sains	B1007
Sains	B1007	Sejarah	B1004	Sejarah	B1004
Fiksi	**B1008**	Teknologi	B1005	Teknologi	B1005

The four Fiksi books keep the order B1001, B1002, B1003, B1008 only in the stable sort.

Figure 3.14 Sorting the catalog by category. Only the stable sort keeps the four *Fiksi* books in their original id order; selection sort moves B1001 behind B1002 and B1003.

Figure 3.14: sorted by category, insertion sort keeps the four Fiksi books in id order; selection sort's first exchange throws B1001 behind B1002 and B1003. To sort by several keys, sort by the least important key first, then stably by the more important ones. Book: Section 3.12.1.

The Complete Table

Table 3.2 Running times (each entry is Θ of the expression, except Shell sort's average, which is an empirical estimate), extra memory, and the three properties of every sort in this chapter.

Algorithm	Best	Average	Worst	Extra space	Stable	In-place	Adaptive
Bubble (early exit)	n	n^2	n^2	1	yes	yes	yes
Exchange	n^2	n^2	n^2	1	no	yes	no
Selection	n^2	n^2	n^2	1	no	yes	no
Insertion	n	n^2	n^2	1	yes	yes	yes
Shell (Knuth gaps)	$n \lg n$	about $n^{1.25}$	$n^{1.5}$	1	no	yes	yes
Merge	$n \lg n$	$n \lg n$	$n \lg n$	n	yes	no	no
Quick (random pivot)	$n \lg n$	$n \lg n$	n^2	$\lg n$	no	yes	no
Heap	$n \lg n$	$n \lg n$	$n \lg n$	1	no	yes	no
Counting	$n+k$	$n+k$	$n+k$	$n+k$	yes	no	no
Radix (d digits, base b)	$d(n+b)$	$d(n+b)$	$d(n+b)$	$n+b$	yes	no	no

Table 3.2: quicksort's extra space is its recursion stack; Shell sort's average is an empirical estimate. Book: Section 3.12.2.

The Same Six Numbers, Very Different Work

Table 3.3 The work each algorithm did on $A = [22, 10, 15, 3, 8, 2]$, as printed by the programs of Sections 3.3 to 3.11.

Algorithm	Comparisons	Swaps or moves
Bubble sort	15	13 swaps
Exchange sort	15	13 swaps
Selection sort	15	3 swaps
Insertion sort	15	13 shifts
Shell sort (gaps 3, 1)	11	7 shifts
Merge sort	9	16 moves
Quicksort, Lomuto, last pivot	13	3 swaps
Quicksort, Hoare, middle pivot	24	3 swaps
Heap sort	13	9 swaps
Counting / radix sort	0	6 / 12 moves

Table 3.3: Hoare's 24 counts every test of its scanning loops, including the ones that stop them; on six elements such differences are noise. To see which sorts are fast, we must go large. Book: Section 3.12.2.

Measured Times

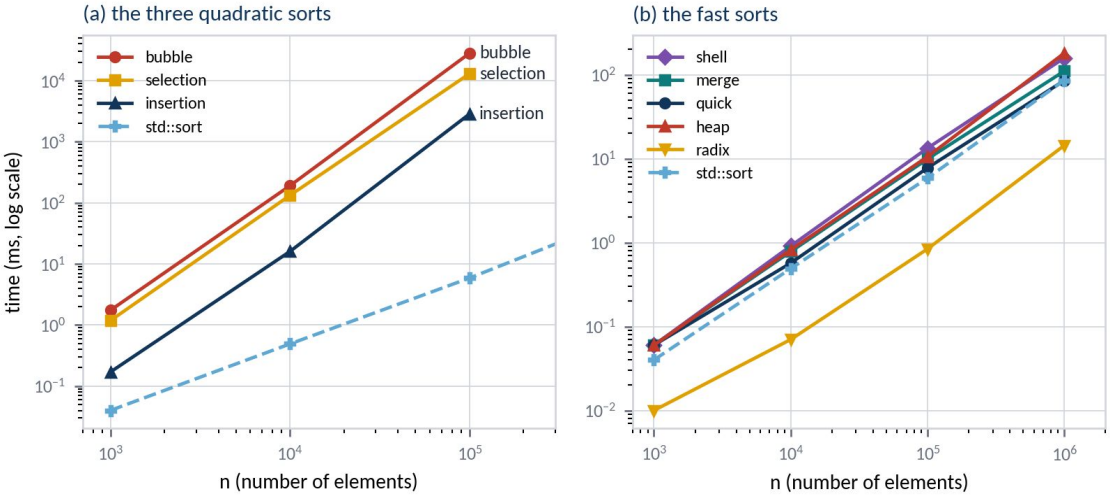


Figure 3.15 Measured times of the C++ programs, on log–log axes. (a) The three quadratic sorts climb two decades of time for each decade of n; `std::sort` is shown for scale. (b) The fast sorts climb only slightly more than one decade per decade; radix sort is fastest.

Figure 3.15: on log–log axes a quadratic sort climbs two decades of time per decade of n; at 10^5 elements bubble sort took 28 seconds and `std::sort` 6 milliseconds. On sorted input bubble and insertion sort finish in microseconds; selection sort takes the same time whatever the order. Book: Sections 3.13.1–3.13.3.

Comparisons Are Not the Whole Story

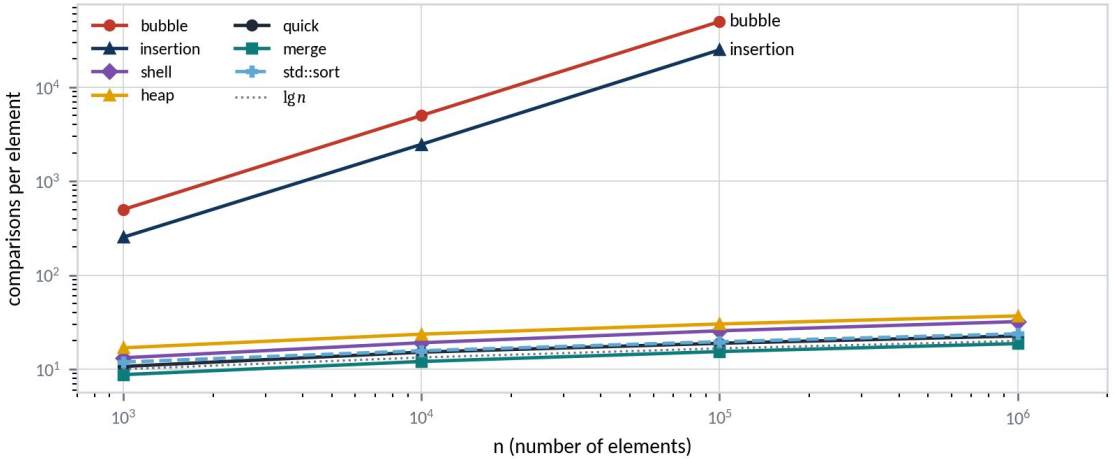


Figure 3.16 Comparisons per element. The quadratic sorts climb steeply; the others stay close to $\lg n$, with merge sort the most economical and heap sort the least.

Figure 3.16: merge sort compares least, heap sort most; yet quicksort and `std::sort` *were fastest at* 10^6 *(about 85 ms), because they move less data and scan memory in order. Radix sort, with no comparisons, took 14 ms. Book: Section 3.13.2.*

Library Sorts Today

Table 3.4 The standard sorts of C++, Java and Python in September 2026, the algorithms inside them, and their guarantees.

Language and call	Algorithm	Stable	Worst case
C++ `std::sort`	introsort: median-of-three quicksort, heap sort when too deep, insertion sort on small pieces	no	$O(n \lg n)$
C++ `std::stable_sort`	merge sort (insertion sort on small runs)	yes	$O(n \lg n)$ with a buffer, $O(n \lg^2 n)$ without
C++ `std::partial_sort`	heap-based selection of the smallest k	no	$O(n \lg k)$
Java `Arrays.sort(int[])` and other primitives	dual-pivot quicksort with insertion sort, run merging, heap sort fallback; counting sort for `byte`, `short`, `char`	not needed	$O(n \lg n)$
Java `Arrays.sort(Object[])`, `List.sort`	TimSort	yes	$O(n \lg n)$
Python `list.sort`, `sorted`	Timsort with the Powersort merge policy (since 3.11)	yes	$O(n \lg n)$

Table 3.4: `std::sort` is introsort (quicksort, heap sort, insertion sort); `std::stable_sort` is merge sort; Java sorts primitives with dual-pivot quicksort and objects with TimSort; Python uses Timsort with the Powersort merge policy since 3.11. Use them, and put your effort into the comparator or key. Book: Section 3.14.

Profitina in Practice

The signals list is sorted by impact, highest first; a *stable* sort keeps equal impacts in a fixed order, so the page does not reshuffle on every visit.

The weekly trend arrives almost sorted by time: Timsort finds the long run and merges the stragglers in close to n comparisons.

Leaderboards need only the top few: `heapq.nlargest` or `std::partial_sort`, $O(n \lg k)$.

PostgreSQL chooses quicksort, top-N heapsort or external merge for `ORDER BY` — the three $n \lg n$ sorts of today.

Reading `EXPLAIN ANALYZE` with Table 3.2 in mind shows when an index would let the database skip the sort. Book: Section 3.15.

Profitina Laundry: Heaviest First

Listing 3.8 From `laundry_orders.py`: the washing queue, heaviest first, with a stable sort.

```
# stable, even reversed
by_weight = sorted(ORDERS, key=lambda o: o[3], reverse=True)
print("Washing queue, heaviest first (equal weights keep arrival order):")
for o in by_weight:
    print("  #%d %s %-5s %4.1f kg" % o)
```

```
Washing queue, heaviest first (equal weights keep arrival order):
  #1008 11:50 Tono   8.0 kg
  #1004 09:30 Andi   7.5 kg
  #1006 10:40 Joko   6.0 kg
  #1010 13:05 Agus   5.5 kg
  #1002 08:10 Budi   5.0 kg
```

Listing 3.8: Maya's and Hadi's orders both weigh 3.0 kg; Maya's, which arrived first, stays first, because `sorted` is stable even with `reverse=True`. C++ needs `std::stable_sort` for the same guarantee; Java's `List.sort` is stable. Book: Section 3.15.

Common Mistakes

A swap without a temporary, or a swap function that receives values; an inner loop that reads `a[n]`.

Testing the element before the index; a `swapped` flag reset only once.

Losing stability with one character (`>=` instead of `>`); a merge buffer allocated in every merge; `(lo + hi) / 2`.

Mixing Lomuto's and Hoare's recursive calls; a first or last pivot on data you do not control.

A comparator with `<=`; counting in `int`; believing quadratic is fine; writing your own sort in production.

Fifteen mistakes, each with its fix, are listed in the book. Book: Section 3.16.

Summary: Five Things to Remember

1. Sorting is a rearrangement into order (Equation (3.1)); inversions (Equation (3.2)) measure how unsorted an array is.

2. Adjacent-swap sorts need $n(n-1)/4$ swaps on average (Equation (3.5)); insertion sort is the best of them: stable, adaptive, $n+inv(A)$.

3. Merge sort is $\Theta(n \lg n)$ always (Equation (3.9)) but needs a buffer; quicksort is fastest in practice but $\Theta(n^2)$ with a bad pivot (Equation (3.10)).

4. Heap sort is $O(n \lg n)$ worst case and in-place; no comparison sort beats $\Omega(n \lg n)$ (Equation (3.13)); counting and radix sort do, on integer keys.

5. Stability enables multi-key sorting; library sorts are hybrids — use them.

Better algorithms beat better hardware: a quarter-century of hardware bought a factor of 10 to 60, while $n \lg n$ against n^2 at a million elements is a factor of 25,000. Book: Section 3.17.

Next Time

Chapter 4 — Evaluation and Discussion I. Worked problems on Chapters 1 to 3, several asking us to sort strings and records with our own comparator.

Every problem goes from the statement to a design, to code in three languages, to a check.

Before then: work Exercises 3.1, 3.3, 3.9, 3.12 and 3.17 by hand, and 3.4 in code.

Exercise 3.17 is the multi-key rule of stability applied to Profitina Laundry. Book: Sections 3.18 and 3.19.

Lecture 4 — Evaluation and Discussion I: Arrays, Searching and Sorting, Solved Step by Step

Course: Data Structure (EF234201)

Book: Irfan Subakti, *Data Structure with C++, Java and Python*, Chapter 4

Lecturer: Irfan Subakti — yifana@gmail.com

No new data structure today: we work eleven problems on Chapters 1 to 3, each from a blank page, in C++, Java and Python, and each checked against a hand trace. Book: Chapter 4, Sections 4.1–4.17.

What This Meeting Covers

1. **A method** — eight steps from a statement to a checked program, and a budget for estimating cost before coding.
2. **Four classic problems** — sorting words, the first repeated character, money by name, sorting by importance.
3. **Seven new problems** — array addresses, counting a key, interpolation search, tracing the elementary sorts, merging, deleting in place, two prices that match a voucher.
4. **Looking back** — the problems side by side, Profitina, the common mistakes.

Every problem runs the same route: the statement, understood by hand, designed on paper, solved in C++ (and Java and Python), a tempting wrong solution, and a check-yourself variant. Book: Section 4.2.3.

Where We Stand: the Toolkit

Table 4.1 The eleven problems of this chapter and the tools from Chapters 1 to 3 that each one exercises most.

Problem	Topic	Main tools from Chapters 1–3
4.1	Sorting words alphabetically	insertion sort, a comparison rule for strings
4.2	The first repeated character	an array of counters indexed by character, two passes
4.3	Looking up money by name	an array of records, sequential search
4.4	Sorting by importance	a two-key comparison rule, stability
4.5	Array addresses by hand	one- and two-dimensional arrays in memory
4.6	Counting a key in a sorted array	binary search for a boundary
4.7	Interpolation search on book codes	interpolation search, comparison with binary search
4.8	Tracing the elementary sorts	bubble, selection and insertion sort, counting steps
4.9	Merging two sorted lists	the merge step of merge sort
4.10	Deleting a value in place	array deletion, a read index and a write index
4.11	Two prices that match a voucher	sorting, then a two-index walk

***Table 4.1**: Chapter 1 gave arrays in memory, Chapter 2 sequential, sentinel, binary and interpolation search, Chapter 3 the sorts and the word stable. Unsure of binary search? Start with Problems 4.3 and 4.6. Book: Section 4.1.*

How to Approach a Problem

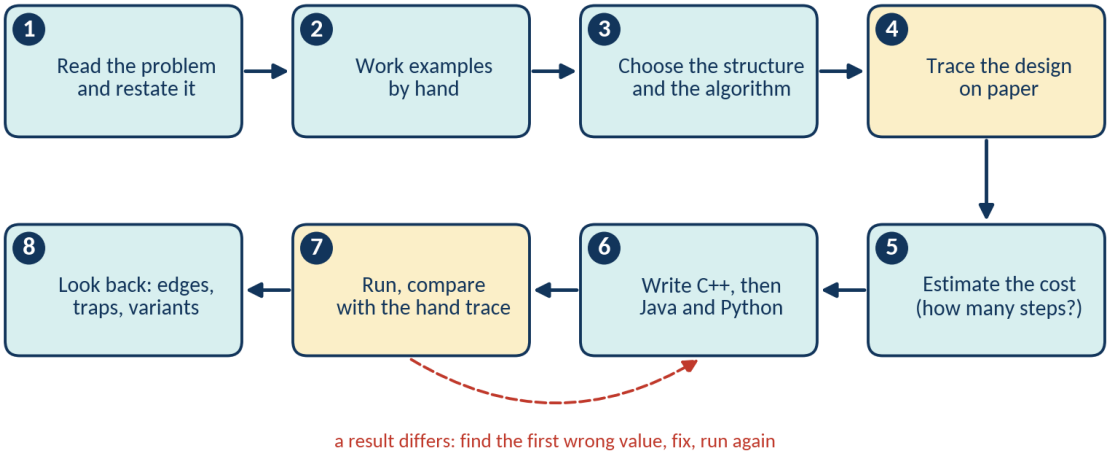


Figure 4.1 The eight steps used for every problem in this chapter; the dashed arrow is the loop between running the program and fixing it.

***Figure 4.1**: Pólya's four movements — understand, plan, carry out, look back — stretched into eight small steps; the dashed arrow is the loop between running the program and fixing it. Book: Section 4.2.*

The Eight Steps

1. Restate what goes in and what comes out. **2. Work examples by hand**, including an edge of your own: empty, one item, missing key, first or last position, ties.

3. Choose the structure and the algorithm — most problems are a known algorithm in disguise. **4. Trace the design on paper**, one line per step.

5. Estimate the cost. 6. Write the code, C++ first, in small tested pieces, then Java and Python.

7. Run and compare with the hand trace: find the *first* value that differs. **8. Look back**: untried inputs, the tempting wrong solution, a variant.

> **WHY THE TRACE COMES BEFORE THE CODE.** A program can only be as right as your understanding of the problem. If you cannot move the indices through nine boxes on paper, you cannot check a program that claims to.

Every expected output today was first computed by hand. Book: Section 4.2.1.

A Budget Before the Code

Table 4.2 Rough step counts for the growth rates of Chapters 2 and 3; at about 10^8 to 10^9 simple steps per second, 10^{12} steps take many minutes and 10^{16} steps take years.

Steps needed	n=100	n=10^4	n=10^6	n=10^8
$\log_2 n$ (binary search)	7	14	20	27
n (one pass)	10^2	10^4	10^6	10^8
$n\log_2 n$ (merge sort)	about 700	about $1.3 \cdot 10^5$	about $2 \cdot 10^7$	about $2.7 \cdot 10^9$
n^2 (two nested loops)	10^4	10^8	10^{12}	10^{16}

$$(n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2} \tag{4.1}$$

$$\lfloor \log_2 n \rfloor + 1 \tag{4.2}$$

***Table 4.2**: about 10^8 to 10^9 simple steps a second. Two nested loops over pairs cost Equation (4.1) — 21 for $n = 7$, 28 for $n = 8$; binary search needs at most Equation (4.2) probes — 4 for eight codes, 20 for a million. Book: Section 4.2.2.*

Problem 4.1: Sorting Words Alphabetically

PROBLEM 4.1. Create a program that sorts the words in alphabetical order.

A one-sentence statement: *we* decide the input (n, then n words), the output, and the order — dictionary order, with the capitalized word first on a tie.

Input `mango Banana apple Cherry durian banana Apple grape`; by hand: `Apple apple Banana banana Cherry durian grape mango`.

Characters are compared by **code**: `A`–`Z` are 65–90 and `a`–`z` are 97–122, so plain `<` says `"Cherry" < "apple"`.

Short statements are the most dangerous: every decision the statement does not make, you must make yourself, and write down. Book: Sections 4.3.1 and 4.3.2.

Problem 4.1: the Sort Stays, the Comparison Changes

Pseudocode 4.1 Insertion sort of words; the comparison COMES-BEFORE implements dictionary order.

```
INSERTION-SORT-WORDS(words, n)
1  for i = 1 to n - 1
2      key = words[i]
3      j = i - 1
4      while j >= 0 and COMES-BEFORE(key, words[j])
5          words[j + 1] = words[j]        // shift one place to the right
6          j = j - 1
7      words[j + 1] = key
```

`COMES-BEFORE(a, b)`: compare small-letter copies; if they differ, that decides; if they tie, `a < b` in code order puts `Apple` before `apple`.

Pseudocode 4.1 *is the insertion sort of Chapter 3 with one new name in line 4. Insertion sort is short, stable (Problem 4.4 needs that) and fast on nearly sorted lists; the cost is* $O(n^2 L)$ *for words of L letters. Book: Section 4.3.3.*

Problem 4.1: the Trace

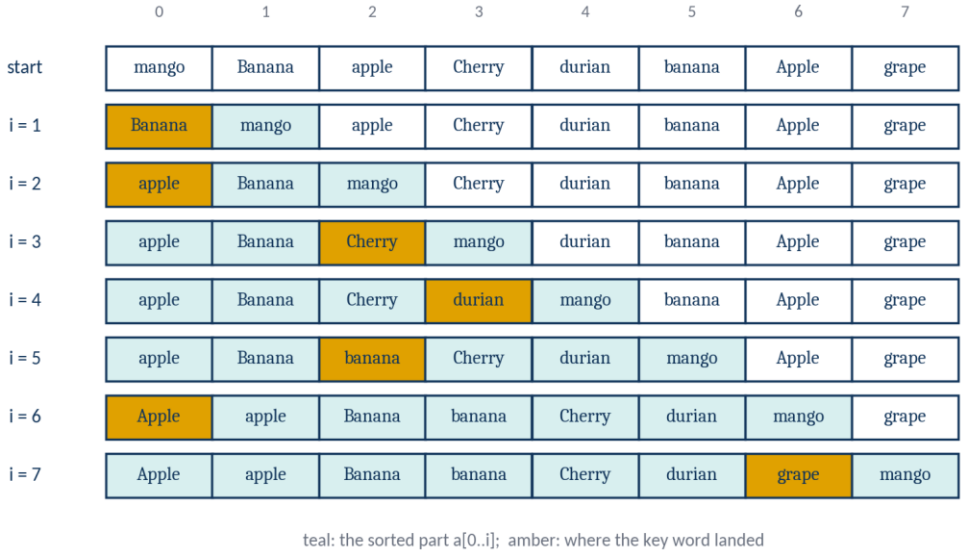


Figure 4.2 Insertion sort of the eight words: after step i the first $i + 1$ words (teal) are in dictionary order, and the amber cell is where the key word landed.

***Figure 4.2**: in row $i = 3$ the key `Cherry` passes `mango` and stops at `Banana`; in row 6 the key `Apple` travels to index 0, past `apple`. Book: Section 4.3.3.*

Problem 4.1: the Solution

Listing 4.1 From `sort_words.cpp`: the dictionary-order comparison.

```
// True when word a must stand before word b in the dictionary.
bool comesBefore(const string& a, const string& b) {
    string la = lowerCopy(a), lb = lowerCopy(b);
    if (la != lb) {
        return la < lb;          // different words: dictionary order
    }
    return a < b;                // same letters: capital first
}
```

```
Sorted:
Apple
apple
Banana
banana
Cherry
durian
grape
mango
```

Listing 4.1*: the loop `j >= 0 && comesBefore(key, words[j])` relies on short-circuit evaluation. Java uses `compareTo` and `equals`; Python `a.lower()`; the library route is `sorted(words, key=lambda w: (w.lower(), w))`. Book: Sections 4.3.4 and 4.3.5.*

Problem 4.1: the Tempting Wrong Solution

Keep the sort, compare with the plain `<`: the output is sorted — by **character code**.

```
Apple
Banana
Cherry
apple
banana
durian
grape
mango
```

Cherry stands before *apple*. On an input written entirely in small letters, the error is invisible.

> **CHECK YOURSELF.** Reverse dictionary order: only `comesBefore` changes (`la > lb`, and `a > b` on a tie); the list starts with `mango` and ends `apple`, `Apple`.

`wrong_sort_words.cpp` is why step 2 asks for an example of your own, and why ours mixes capitals on purpose. Book: Section 4.3.6.

Problem 4.2: the First Repeated Character

> **PROBLEM 4.2.** Find the first duplicate character in a string. Example 1: `abbcbbca` gives `a`. Example 2: `abbcc` gives `b`. The optimal solution is $O(n)$.

Reading A: stop at the first character already seen — `b` for Example 1. **Reading B**: the leftmost character that occurs more than once — `a`.

Example 1 says `a`: Reading B. Example 2 alone could not tell them apart. Check *every* example.

Own examples: `abcdef` (none), `Profitina` (`i`), `hello world` (`l`), `xyzzyx` (`x` under B, `z` under A).

Restated: count every character; the answer is the leftmost character whose count is at least 2. Book: Sections 4.4.1 and 4.4.2.

Problem 4.2: the Value Is the Index

Pseudocode 4.2 The first repeated character in two passes, with one counter per possible character.

```
FIRST-REPEATED(s)
1  let count[0..255] be a new array, all zeros
2  for i = 0 to length(s) - 1          // pass 1: count
3      count[s[i]] = count[s[i]] + 1
4  for i = 0 to length(s) - 1          // pass 2: leftmost with count >= 2
5      if count[s[i]] >= 2
6          return i
7  return -1                             // no character repeats
```

***Pseudocode 4.2**: all pairs would cost Equation (4.1), about $5 \cdot 10^{11}$ for a million characters. Instead, 256 counters, one per possible byte, indexed by the character itself: `count['a']` is `count[97]`. Two passes, $2n + 256$ steps: $O(n)$ time, $O(1)$ extra space. This is the idea of the hash table of Chapter 15. Book: Section 4.4.3.*

Problem 4.2: the Trace

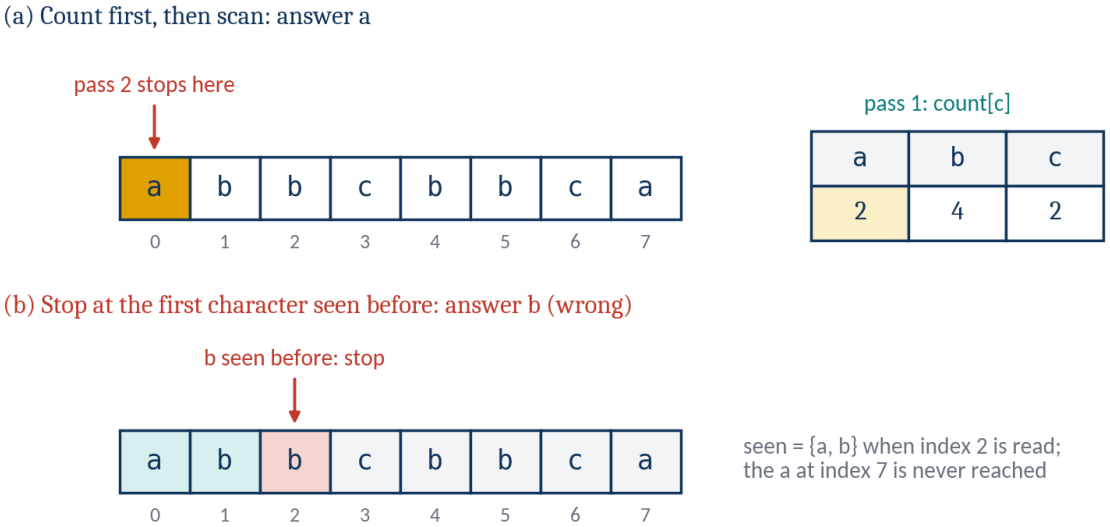


Figure 4.3 (a) Pass 1 fills the counters, pass 2 stops at the first character whose counter is at least 2; (b) stopping at the first character seen before gives the wrong answer b.

***Figure 4.3**: (a) pass 1 leaves `count['a'] = 2`, `count['b'] = 4`, `count['c'] = 2`, and pass 2 stops at once on `a`; (b) Reading A stops at the second `b`, the tempting wrong answer. Book: Section 4.4.3.*

Problem 4.2: the Solution

Listing 4.2 From `first_repeat.cpp`: two passes with 256 counters.

```
// Returns the position of the first character that occurs at least twice,
// or -1 when every character occurs only once.
int firstRepeated(const string& s) {
    int count[256] = {0};                  // one counter per possible char
    for (size_t i = 0; i < s.size(); i++) {  // pass 1: count them all
        count[(unsigned char)s[i]]++;
    }
    for (size_t i = 0; i < s.size(); i++) {  // pass 2: first count >= 2
        if (count[(unsigned char)s[i]] >= 2) {
            return (int)i;
        }
    }
    return -1;
}
```

```
abbcbbca -> a
abbcc -> b
xyzzyx -> x
```

***Listing 4.2**: `= {0}` zeroes the counters; `unsigned char` keeps every index in 0..255. The one-pass Reading A answers `b` for `abbcbbca` and `z` for `xyzzyx`. Book: Sections 4.4.4 and 4.4.6.*

Problem 4.3: Looking Up Money by Name

> **PROBLEM 4.3.** Search the money owned by persons; print the amount, or "Not found". Data: Puspita 50, Bejo 35, Sarita 20, Pujo 45, Lestari 60. `Pujo` gives 45; `Komeng` gives Not found.

Own queries: `Lestari` (last row, worst case, 5 comparisons), `Puspita` (first row, 1), `pujo` (exact match: Not found).

Structure: one `struct Person { string name; int money; }` per row, so a name can never be separated from its amount.

Algorithm: the names are not sorted, so sequential search, **Pseudocode 4.3**.

Pseudocode 4.3 Sequential search over an array of records; the result is an index, or −1 when the name is absent.

```
SEQUENTIAL-SEARCH(people, n, name)
1  for i = 0 to n - 1
2      if people[i].name == name
3          return i                  // found: stop at once
4  return -1                         // all n rows looked at: not there
```

$O(n)$ per query; for a million customers and a million queries a day, sort once and binary-search. Book: Sections 4.5.1–4.5.3.

Problem 4.3: Two Searches, One Unsorted Array

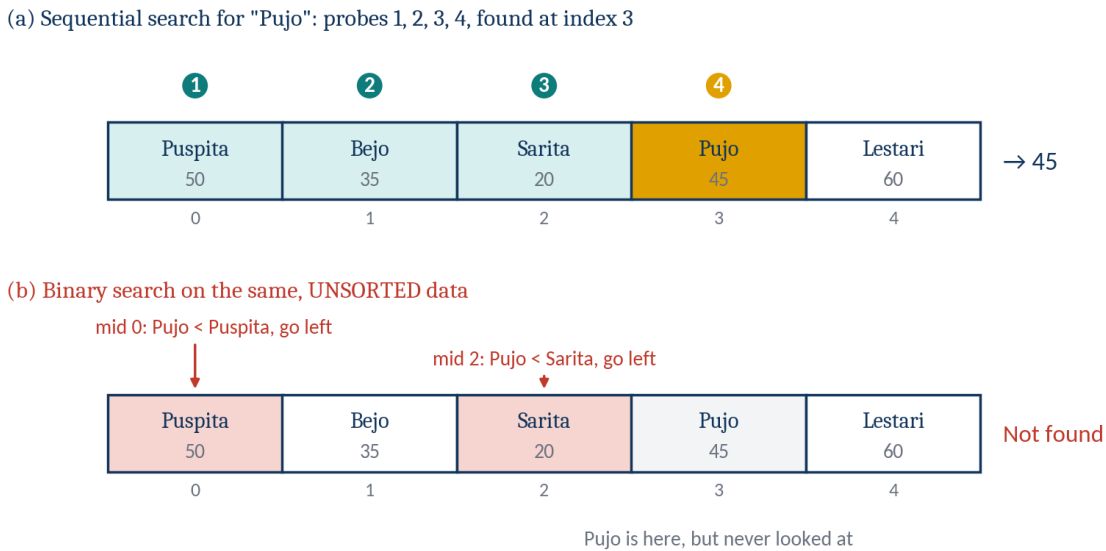


Figure 4.4 (a) Sequential search finds Pujo with its fourth probe; (b) binary search on the same unsorted data discards the part that holds Pujo and reports Not found.

***Figure 4.4**: (a) sequential search finds Pujo with its fourth probe; (b) binary search probes Sarita, concludes that Pujo lies to the left, and throws away the very box that holds it. Book: Sections 4.5.3 and 4.5.6.*

Problem 4.3: the Solution

Listing 4.3 From `search_money.cpp`: sequential search over an array of `Person` records.

```
// Returns the index of the person called name, or -1 if there is none.
int sequentialSearch(const Person people[], int n, const string& name) {
    for (int i = 0; i < n; i++) {
        if (people[i].name == name) {
            return i;                       // found: stop at once
        }
    }
    return -1;                              // looked at all n: not there
}
```

```
45
Not found
60
Not found
50
```

> **JAVA STRINGS.** `people[i].name == name` compares *references* in Java; a name read from the keyboard is a new object. Use `equals`.

Listing 4.3 *on `Pujo`, `Komeng`, `Lestari`, `pujo`, `Puspita`. The binary-search version prints Not found four times: a precondition that is not met gives no error, only wrong answers. Book: Sections 4.5.4–4.5.6.*

Problem 4.4: Sorting by Importance

PROBLEM 4.4. Sort strings from most to least important: more `'!'` is more important; with equal `'!'`, the longer string is less important.

Table 4.3 The two keys of every string of Problem 4.4, counted by hand.

String	'!'	Length
`!important`	1	9
`not important at all`	0	20
`meh`	0	3
`Most Important!!!!`	4	18
`how about this?`	0	15
`important!`	1	10

***Table 4.3**: three groups by the first key — 4 marks, then 1, then 0 — and by length inside each group. Two strings that tie on both keys keep their input order: the sort must be **stable**. Book: Sections 4.6.1 and 4.6.2.*

Problem 4.4: Two Keys in One Rule

Pseudocode 4.4 A two-key comparison rule and the stable insertion sort that uses it.

```
MORE-IMPORTANT(a, b)
1  ca = number of '!' in a;  cb = number of '!' in b
2  if ca != cb
3      return ca > cb                  // first key: more marks first
4  return length(a) < length(b)        // second key: shorter first (strict)

SORT-BY-IMPORTANCE(lines, n)
5  INSERTION-SORT-WORDS(lines, n) with COMES-BEFORE replaced by MORE-IMPORTANT
```

Listing 4.4 From `importance_sort.cpp`: the two-key comparison.

```cpp
// True when a is MORE important than b, so a must come first.
bool moreImportant(const string& a, const string& b) {
    int ca = countBangs(a), cb = countBangs(b);
    if (ca != cb) {
        return ca > cb;                     // first key: more '!' first
    }
    return a.size() < b.size();             // second key: shorter first
}
```

Pseudocode 4.4 *and* ***Listing 4.4****: the length test is strict, so a tie on both keys answers "no" both ways and insertion sort leaves the pair in place. Book: Sections 4.6.3 and 4.6.4.*

Problem 4.4: the Result

Input order

string	'!'	length
!important	1	9
not important at all	0	20
meh	0	3
Most Important!!!!	4	18
how about this?	0	15
important!	1	10

Most to least important

string	'!'	length
Most Important!!!!	4	18
!important	1	9
important!	1	10
meh	0	3
how about this?	0	15
not important at all	0	20

first key: more '!' first; second key (ties only): shorter first

Figure 4.5 The six strings with their two keys, in input order (left) and sorted from most to least important (right); the colors mark the three groups of the first key.

Figure 4.5: `Most Important!!!!`, `!important`, `important!`, `meh`, `how about this?`, `not important at all` — *exactly the output of the statement. In Python:* `sorted(lines, key=lambda s: (-s.count("!"), len(s)))`. *Book: Sections 4.6.3–4.6.5.*

Problem 4.4: Two Stable Sorts, Wrong Order

Listing 4.5 From `wrong_importance_sort.py`: two stable sorts, applied in the wrong order.

```
lines = [line.rstrip("\r\n") for line in sys.stdin]
lines.sort(key=lambda s: -s.count("!"))    # 1st: by number of '!' ...
lines.sort(key=len)                         # 2nd: ... then by length (wins!)
for s in lines:
    print(s)
```

```
meh
!mportant
important!
how about this?
Most Important!!!!
not important at all
```

A stable sort keeps the previous order only *among ties of the current key*: the **last** sort decides the main order.

Rule: **sort by the least important key first, the most important last**.

> **CHECK YOURSELF.** Add `Hi!` and `Yo!`: both have one mark and length 3, so the output starts `Most Important!!!!`, `Hi!`, `Yo!`, `!mportant` — `Hi!` first, by stability.

Listing 4.5 *sorts by length last, so length wins. Book: Section 4.6.6.*

Problem 4.5: Array Addresses by Hand

PROBLEM 4.5. `int A[10]` and `int B[3][5]` both start at `0x1000`. Find the addresses of `A[7]` and `B[2][3]` for 4-byte and 2-byte `int`s, and check against real memory.

`A[7]`: $7 \cdot 4 = 28$ bytes past the base, `0x101C`; with 2 bytes, `0x100E`.

`B[2][3]`: row-major, two full rows of 5 and then 3: 13 cells, 52 bytes, `0x1034`; with 2 bytes, `0x101A`.

$$\text{addr}(A[i]) = b + i \cdot s \tag{4.3}$$

$$\text{addr}(B[r][c]) = b + (r \cdot C + c) \cdot s \tag{4.4}$$

Equation (4.3) and Equation (4.4): the number of rows never enters the address. Book: Sections 4.7.1–4.7.3.

Problem 4.5: Memory, Drawn

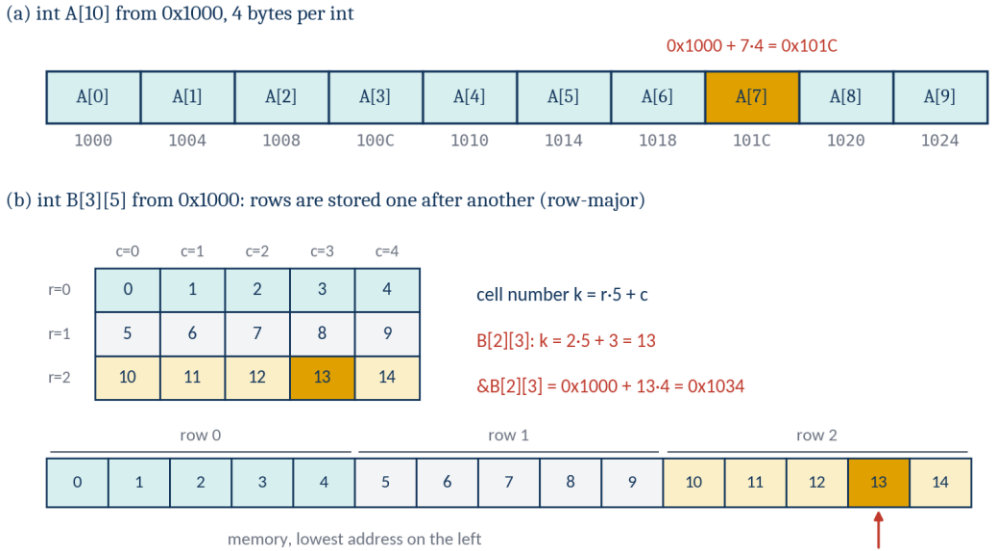


Figure 4.6 (a) The ten boxes of A and their addresses; (b) the grid B, its cells numbered in memory order, and the same cells as they really lie in memory.

Figure 4.6*: (b) numbers the cells of* `B` *in memory order, so* $k = r \cdot 5 + c$ *can be read off. Book: Section 4.7.3.*

Problem 4.5: the Formulas, Checked

Pseudocode 4.5 The address of an array element in one and two dimensions (row-major order).

```
ADDRESS-1D(b, s, i)
1  return b + i * s
ADDRESS-2D(b, s, C, r, c)
2  return b + (r * C + c) * s        // C = number of columns
```

```
int of 4 bytes, both arrays starting at 0x1000
  &A[7]    = 0x1000 + 7 * 4 = 0x101C
  &B[2][3] = 0x1000 + (2 * 5 + 3) * 4 = 0x1034
int of 2 bytes, both arrays starting at 0x1000
  &A[7]    = 0x1000 + 7 * 2 = 0x100E
  &B[2][3] = 0x1000 + (2 * 5 + 3) * 2 = 0x101A
On this computer sizeof(int) = 4
  &A[7] - &A[0]       = 28 bytes
  &B[2][3] - &B[0][0] = 52 bytes
```

Pseudocode 4.5 *is constant time — the property of arrays, and the one linked lists give up.* `array_address.cpp` *prints both answers for both sizes and then measures the real distances in memory: 28 and 52 bytes. Book: Sections 4.7.3 and 4.7.4.*

Problem 4.5: Two Slips

Listing 4.6 From `wrong_array_address.py`: forgetting the element size, and multiplying by the number of rows.

```
base, size = 0x1000, 4
# Mistake 1: forgetting that one element is size bytes wide.
print(f"&A[7]    = 0x{base + 7:X}")
# Mistake 2: multiplying the row by the number of ROWS (3), not columns (5).
print(f"&B[2][3] = 0x{base + (2 * 3 + 3) * size:X}")
```

```
&A[7]    = 0x1007
&B[2][3] = 0x1024
```

***Listing 4.6**: `0x1007` is not a multiple of 4 past the base; `0x1024` is less than two full rows (40 bytes) past it. Check yourself: `double M[4][6]` at `0x2000` puts `M[3][5]` at `0x20B8`, and the first byte after the array at `0x20C0`. Book: Sections 4.7.4–4.7.6.*

Problem 4.6: Counting a Key in a Sorted Array

PROBLEM 4.6. Years on one shelf, sorted: 1998, 2001, 2001, 2001, 2005, 2010, 2010, 2019, 2023. How many books from 2001, 2010, 2002, 2023, 1990 and 2030, in $O(\log n)$ time?

Equal values stand together in one block: 2001 occupies indices 1 to 3, so the count is $4 - 1 = 3$.

An absent year has an empty block: 2002 "starts" and ends at 4. 2030's block lies past the end, at index 9.

$$\text{count}(key) = \text{upper}(key) - \text{lower}(key) \tag{4.5}$$

Equation (4.5): the lower bound is the first index with $a[i] \geq key$, the upper bound the first with $a[i] > key$; either may be n. Book: Sections 4.8.1–4.8.3.

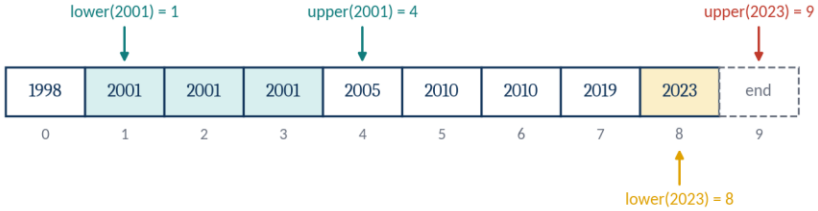
Problem 4.6: Binary Search for a Boundary

Pseudocode 4.6 Binary search for the two boundaries of a block of equal keys in a sorted array.

```
LOWER-BOUND(a, n, key)              // first i with a[i] >= key, or n
1  lo = 0;  hi = n
2  while lo < hi
3      mid = lo + (hi - lo) / 2     // integer division
4      if a[mid] < key
5          lo = mid + 1
6      else hi = mid
7  return lo
UPPER-BOUND(a, n, key)              // first i with a[i] > key, or n
8  the same, with line 4 reading: if a[mid] <= key
```

***Pseudocode 4.6**: the range starts as `[0, n]`, including the past-the-end position, because the answer may be n. `hi = mid` keeps a candidate; `lo < hi` stops when one remains. Book: Section 4.8.3.*

Problem 4.6: the Trace



lowerBound(2001), step by step

lo	hi	mid	a[mid]	decision
0	9	4	2005	hi = mid
0	4	2	2001	hi = mid
0	2	1	2001	hi = mid
0	1	0	1998	lo = mid+1
1	1			lo = hi: stop, answer 1

Figure 4.7 The lower and upper bounds of 2001 and 2023, with the past-the-end position 9 drawn dashed, and a step-by-step trace of LOWER-BOUND(2001).

***Figure 4.7**: `LOWER-BOUND(2001)` shrinks `[0, 9]` to `[0, 4]`, `[0, 2]`, `[0, 1]`, `[1, 1]`: four probes, as Equation (4.2) allows. Book: Section 4.8.3.*

Problem 4.6: the Solution, and the Off-by-One

Listing 4.7 From `count_range.cpp`: the lower bound on the range `[0, n]`.

```cpp
// First index i with a[i] >= key (n if there is none).
int lowerBound(const int a[], int n, int key) {
    int lo = 0, hi = n;                  // the answer lies in [lo, hi]
    while (lo < hi) {
        int mid = lo + (hi - lo) / 2;
        if (a[mid] < key) {
            lo = mid + 1;                // a[mid] too small: go right of mid
        } else {
            hi = mid;                    // a[mid] >= key: mid may be the answer
        }
    }
    return lo;
}
```

```
2001: lower 1, upper 4, count 3
2010: lower 5, upper 7, count 2
2002: lower 4, upper 4, count 0
2023: lower 8, upper 9, count 1
```

***Listing 4.7**: C++ offers `std::lower_bound`, `std::upper_bound` and `std::equal_range`; Python `bisect_left` and `bisect_right`. Starting at `[0, n-1]` can never return 9, so 2023 gets count 0. Check yourself: years 2000–2010 are* $\text{lower}(2011) - \text{lower}(2000) = 7 - 1 = 6$. *Book: Sections 4.8.4–4.8.6.*

Problem 4.7: Interpolation Search on Book Codes

PROBLEM 4.7. Codes 025, 034, 041, 056, 063, 072, 088, 096. Look up 88, 60, 41, 25, 96 and 100 by interpolation search, print every probe, and compare with binary search.

$$pos = low + \left\lfloor \frac{(key - a[low]) \cdot (high - low)}{a[high] - a[low]} \right\rfloor \tag{4.6}$$

88: $\lfloor 63 \cdot 7/71 \rfloor = \lfloor 6.21 \rfloor = 6$, and $a[6] = 88$: one probe.

60: $\lfloor 245/71 \rfloor = 3$, $a[3] = 56 < 60$, `low = 4`; now $a[4] = 63 > 60$: absent after one probe.

41: probe 1 ($a[1] = 34$), then $2 + 0 = 2$: found after two probes.

Equation (4.6): probe the same fraction of the index range as the key is of the value range. Book: Sections 4.9.1 and 4.9.2.

Problem 4.7: Two Guards

Pseudocode 4.7 Interpolation search; the range test in line 2 keeps every probe inside the array, and line 3 prevents a division by zero.

```
INTERPOLATION-SEARCH(a, n, key)
1  low = 0;  high = n - 1
2  while low <= high and a[low] <= key and key <= a[high]
3      if a[high] == a[low]
4          pos = low                        // avoid dividing by zero
5      else pos = low + (key - a[low]) * (high - low) / (a[high] - a[low])
6      if a[pos] == key
7          return pos
8      elseif a[pos] < key
9          low = pos + 1
10     else high = pos - 1
11 return -1
```

Pseudocode 4.7*: line 2's range test keeps the fraction between 0 and 1, so every probe is inside the array; line 3 avoids dividing by zero. About $log_2 log_2 n$ probes on evenly spread keys, but $O(n)$ on skewed ones. Book: Section 4.9.3.*

Problem 4.7: Projecting a Value Onto an Index

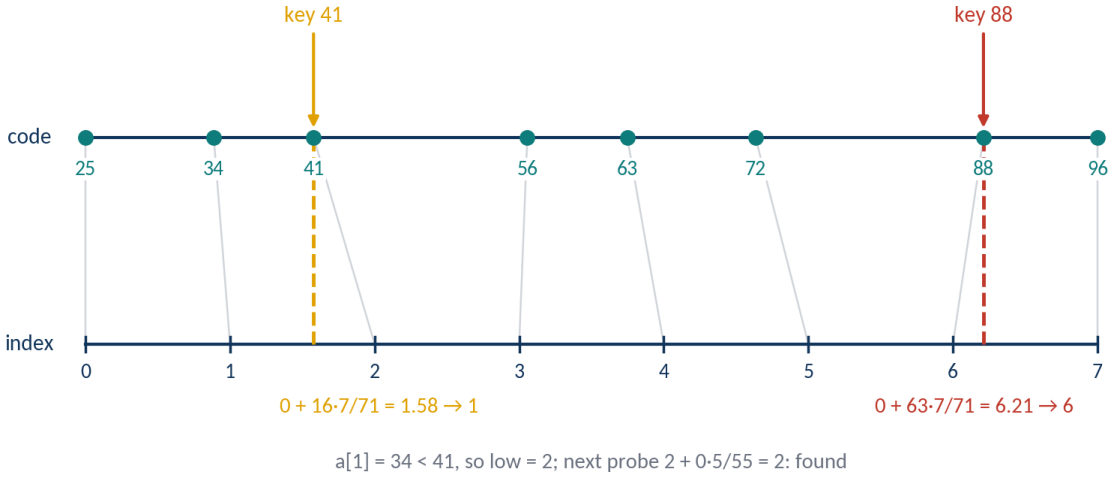


Figure 4.8 Interpolation search projects a key from the axis of code values onto the axis of indices; 88 lands at 6.21 and is found at index 6, while 41 lands at 1.58 and needs a second probe.

***Figure 4.8**: 88 lands at 6.21 and is found at index 6; 41 lands at 1.58 and needs a second probe. Book: Section 4.9.3.*

Problem 4.7: the Solution

Listing 4.8 From `interpolation_search.cpp`: the guarded loop; the product is formed before the division.

```
    // The key can only be here while codes[low] <= key <= codes[high].
    while (low <= high && key >= codes[low] && key <= codes[high]) {
        int pos = low;
        if (codes[high] != codes[low]) {    // avoid dividing by zero
            pos = low + (key - codes[low]) * (high - low)
                      / (codes[high] - codes[low]);
        }
        if (codes[pos] == key) {
            return pos;
        } else if (codes[pos] < key) {
            low = pos + 1;
        } else {
            high = pos - 1;
        }
    }
```

```
Search 88: probe 6 (code 88) -> found at index 6, "Visual Basic 2005 Express"; probes: 1 (binary search: 3)
Search 60: probe 3 (code 56) -> not found; probes: 1 (binary search: 3)
```

***Listing 4.8**: dividing first would almost always give 0. Without the range guard, key 200 probes index 17: `IndexError` in Python, garbage in C++. Book: Sections 4.9.4 and 4.9.6.*

Problem 4.8: Tracing the Elementary Sorts

> **PROBLEM 4.8.** Sort 5, 6, 8, 1, 3, 25, 10 with bubble, selection and insertion sort; write the array after every pass and count comparisons and swaps or shifts.

Count the inversions first: 5, 6 and 8 each beat 1 and 3; 25 beats 10: $2 + 2 + 2 + 1 = 7$.

Bubble sort: 21 comparisons (Equation (4.1)), one swap per inversion: **7**. Selection sort: 21 comparisons, at most 6 swaps. Insertion sort: 7 shifts, and fewer comparisons.

All three are $O(n^2)$ on a reversed array; on a sorted one insertion sort needs only $n - 1$ comparisons. Book: Sections 4.10.1–4.10.3.

Problem 4.8: the Three Sorts

Pseudocode 4.8 Bubble sort, selection sort and insertion sort, as traced in Problem 4.8.

```
BUBBLE-SORT(a, n)
1  for i = 0 to n - 2
2      for j = 0 to n - 2 - i            // the last i places are settled
3          if a[j] > a[j + 1]
4              swap a[j] and a[j + 1]
SELECTION-SORT(a, n)
5  for i = 0 to n - 2
6      m = index of the smallest of a[i..n-1]
7      if m != i
8          swap a[i] and a[m]
INSERTION-SORT(a, n)
9  for i = 1 to n - 1
10     key = a[i];  j = i - 1
11     while j >= 0 and a[j] > key
12         a[j + 1] = a[j];  j = j - 1    // shift
13     a[j + 1] = key
```

Pseudocode 4.8*: the three sorts exactly as Chapter 3 defined them. Book: Section 4.10.3.*

Problem 4.8: Bubble Sort's First Pass

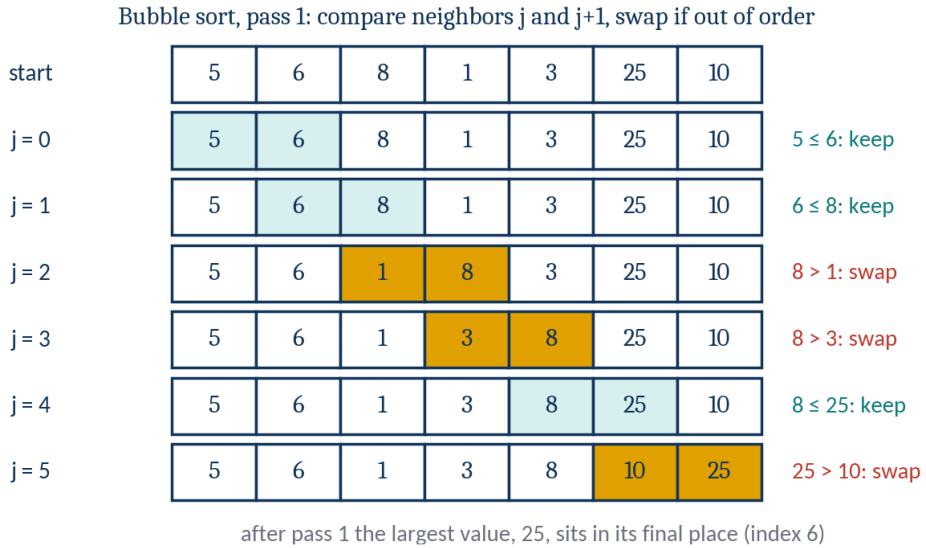


Figure 4.9 The six comparisons of the first pass of bubble sort; amber pairs were swapped, teal pairs were already in order.

Figure 4.9*: six comparisons, three swaps (8 with 1, 8 with 3, 25 with 10); 25 is carried like a bubble to the right end, where it stays. Book: Section 4.10.3.*

Problem 4.8: the Counts

```
Insertion sort
  pass 1: 5 6 8 1 3 25 10
  pass 2: 5 6 8 1 3 25 10
  pass 3: 1 5 6 8 3 25 10
  pass 4: 1 3 5 6 8 25 10
  pass 5: 1 3 5 6 8 25 10
  pass 6: 1 3 5 6 8 10 25
  comparisons 12, shifts 7
```

Table 4.4 The counts of Problem 4.8 for the array 5, 6, 8, 1, 3, 25, 10, which has 7 inversions.

Algorithm	Comparisons	Swaps or shifts	Passes that changed the array
Bubble sort	21	7 swaps	3 of 6
Selection sort	21	5 swaps	5 of 6
Insertion sort	12	7 shifts	3 of 6

***Table 4.4**: every prediction confirmed. Bubble sort finished its real work after pass 3; a `swapped` flag would have stopped after pass 4. Check yourself: on the reversed array insertion sort makes 21 comparisons and 21 shifts; on the sorted one, 6 and 0. Book: Section 4.10.4.*

Problem 4.8: One Step Too Far

Listing 4.9 From `WrongBubble.java`: the inner loop runs one step too far.

```
        for (int i = 0; i < n - 1; i++) {
            for (int j = 0; j < n - i; j++) {        // should be j < n - 1 - i
                if (a[j] > a[j + 1]) {
                    int t = a[j]; a[j] = a[j + 1]; a[j + 1] = t;
                }
            }
        }
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 7 out of bounds for length 7
    at WrongBubble.main(WrongBubble.java:9)
```

***Listing 4.9**: at $i = 0$ the last `j` is 6, and `a[j + 1]` reads `a[7]`. Java names the bad index, the length and the line — a gift. The same bug in C++ reads and may write memory it does not own, silently; hence `-fsanitize=address`. Book: Section 4.10.6.*

Problem 4.9: Merging Two Sorted Lists

PROBLEM 4.9. Branch A: 3, 9, 12, 15, 31. Branch B: 5, 9, 17, 23, 35, 40, 41. Produce one sorted catalog in one pass, and count the comparisons.

Two face-up piles: take the smaller top card. On the tie 9–9, take A's first (stable).

After 31, A is empty: nine comparisons, and 35, 40, 41 are copied without any.

Pseudocode 4.9 Merging two sorted arrays into a third; at most one of the two copy loops does any work.

```
MERGE(a, n, b, m, c)
1  i = 0;  j = 0;  k = 0
2  while i < n and j < m            // both arrays still have items
3      if a[i] <= b[j]              // <= takes a tie from a: stable
4          c[k] = a[i];  i = i + 1
5      else c[k] = b[j];  j = j + 1
6      k = k + 1
7  while i < n                      // copy the rest of a, if any
8      c[k] = a[i];  i = i + 1;  k = k + 1
9  while j < m                      // copy the rest of b, if any
10     c[k] = b[j];  j = j + 1;  k = k + 1
```

***Pseudocode 4.9**: every element taken is the smallest not yet taken, so C grows in order. At most $n + m - 1$ comparisons: $O(n + m)$. Book: Sections 4.11.1–4.11.3.*

Problem 4.9: Where Every Element Came From

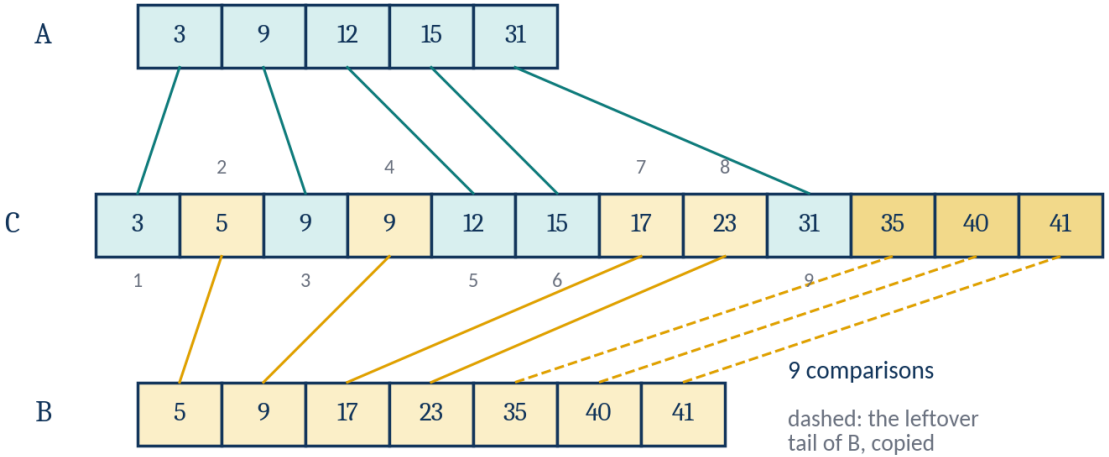


Figure 4.10 Merging A and B into C: each element of C is joined to its source, the small numbers count the comparisons, and the dashed tail of B is copied without comparisons.

Figure 4.10*: teal lines from A, amber from B, dashed for the tail of B copied without comparisons. Sorting the twelve codes from scratch could cost 66 comparisons. Book: Section 4.11.3.*

Problem 4.9: the Solution

Listing 4.10 From `merge_sorted.cpp`: the merge, counting comparisons.

```
    while (i < n && j < m) {                // both arrays still have items
        comparisons++;
        if (a[i] <= b[j]) {                 // <= takes a tie from a first
            c[k++] = a[i++];
        } else {
            c[k++] = b[j++];
        }
    }
    while (i < n) {                         // copy what is left of a ...
        c[k++] = a[i++];
    }
    while (j < m) {                         // ... or what is left of b
        c[k++] = b[j++];
    }
```

```
Branch A: 3 9 12 15 31
Branch B: 5 9 17 23 35 40 41
Merged:   3 5 9 9 12 15 17 23 31 35 40 41
Comparisons: 9
```

Listing 4.10: `c[k++] = a[i++]` *copies, then advances both. Leave out the two copy loops and only 9 of 12 codes survive — check that the output has* $n + m$ *elements. Library:* `std::merge`, `heapq.merge`. *Book: Sections 4.11.4–4.11.6.*

Problem 4.10: Deleting a Value in Place

PROBLEM 4.10. Delete every occurrence of x from 7, 3, 3, 5, 3, 9 in place, keeping the order, and report the new size. Test $x = 3$,7,9,4, and 3, 3, 3 with $x = 3$.

An array cannot get shorter: move the kept numbers to the front and remember how many count — **size** versus **capacity**.

Pseudocode 4.10 Deleting every occurrence of x in one pass with a read index r and a write index w.

```
DELETE-ALL(a, n, x)
1  w = 0                              // next free place for a kept item
2  for r = 0 to n - 1                 // read every item exactly once
3      if a[r] != x
4          a[w] = a[r]
5          w = w + 1
6  return w                            // the new size
```

***Pseudocode 4.10**: invariant — `a[0..w-1]` holds, in order, exactly the kept elements of `a[0..r-1]`; since `w <= r`, writing never destroys an unread element. $O(n)$ time, $O(1)$ space, against $O(n^2)$ for repeated shifting. Book: Sections 4.12.1–4.12.3.*

Problem 4.10: Read Index and Write Index

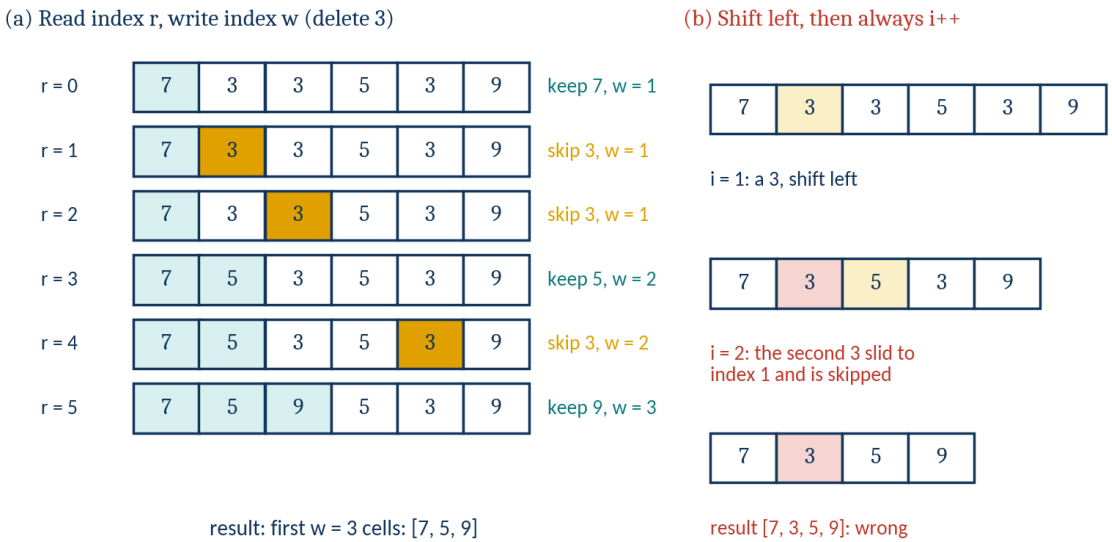


Figure 4.11 (a) The read index r visits every box once and the write index w packs the kept numbers to the front; (b) shifting left and then always advancing i skips the second of two adjacent 3s.

***Figure 4.11**: (a) `r` visits every box once while `w` packs the kept numbers to the front; (b) shifting left and then always advancing skips the second of two adjacent 3s. Book: Section 4.12.3.*

Problem 4.10: the Solution and the Trap

Listing 4.11 From `delete_value.cpp`: one pass with a read index and a write index.

```cpp
// Keeps every item that is not x, packed to the front; returns the new size.
int deleteAll(int a[], int n, int x) {
    int w = 0;                          // next free place for an item we keep
    for (int r = 0; r < n; r++) {       // r reads every item exactly once
        if (a[r] != x) {
            a[w] = a[r];
            w++;
        }
    }
    return w;
}
```

```
delete 3 from [7, 3, 3, 5, 3, 9] -> [7, 5, 9] (3 removed)
delete 7 from [7, 3, 3, 5, 3, 9] -> [3, 3, 5, 3, 9] (1 removed)
[7, 3, 5, 9]
```

Listing 4.11 *is* `std::remove`*, which packs but does not resize. The shift-left version prints* `[7, 3, 5, 9]`*: one 3 survived. Check yourself: removing duplicates from a sorted array keeps* `a[r]` *when* `a[r] != a[w - 1]` *— that is* `std::unique`*. Book: Sections 4.12.4–4.12.6.*

Problem 4.11: Two Prices That Match a Voucher

PROBLEM 4.11. Profitina Laundry's prices: 18000, 7000, 25000, 12000, 30000, 5000, 20000, 15000. For vouchers of 37000, 12000 and 11000, list every pair of different services that matches, in fewer steps than trying all pairs.

All pairs: $8 \cdot 7/2 = 28$. By hand: $7000 + 30000$ and $12000 + 25000$; $5000 + 7000$; none for 11000.

Sort, then **walk inward**: too small, drop the cheapest; too large, drop the dearest.

Pseudocode 4.11 The two-index walk that finds every pair with a given sum in a sorted array.

```
PAIRS-WITH-SUM(a, n, target)            // a must be sorted ascending
1  i = 0;  j = n - 1
2  while i < j
3      s = a[i] + a[j]
4      if s == target
5          report (a[i], a[j]);  i = i + 1;  j = j - 1
6      elseif s < target
7          i = i + 1                    // a[i] is too cheap for any partner
8      else j = j - 1                   // a[j] is too dear for any partner
```

Pseudocode 4.11*: a discarded price cannot belong to any pair with the prices still between the indices, so no pair is missed. $O(n \log n)$ for the sort, $O(n)$ for the walk. Book: Sections 4.13.1–4.13.3.*

Problem 4.11: the Walk for 37000

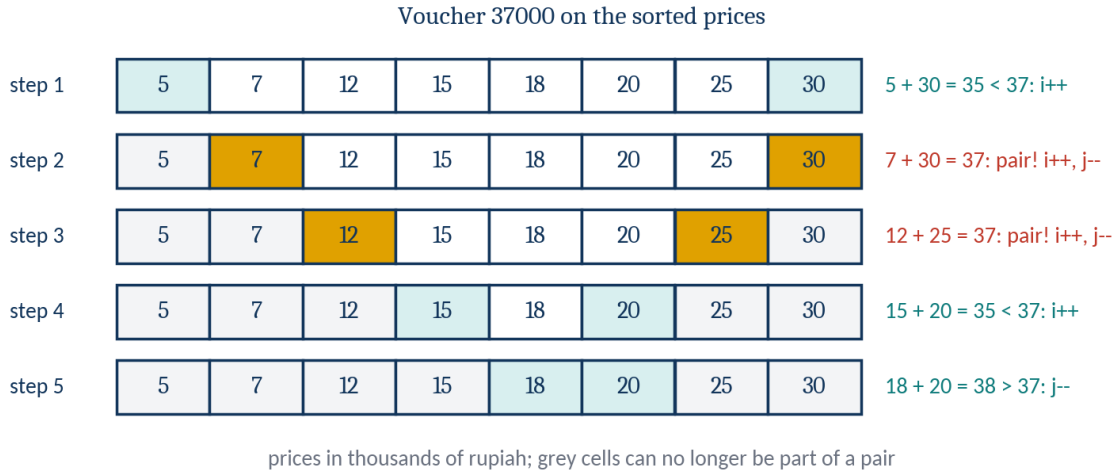


Figure 4.12 The two-index walk for the voucher of 37000 on the sorted prices, in thousands of rupiah: each step either finds a pair or discards the cheapest or the dearest remaining price.

Figure 4.12*: five steps instead of 28 pairs, in thousands of rupiah: each step finds a pair or discards the cheapest or the dearest remaining price. Book: Section 4.13.3.*

Problem 4.11: the Solution

Listing 4.12 From `pair_sum.cpp`: the two-index walk on the sorted prices.

```
    while (i < j) {
        steps++;
        int sum = a[i] + a[j];
        if (sum == target) {
            cout << ' ' << a[i] << " + " << a[j] << ';';
            found++;
            i++;
            j--;
        } else if (sum < target) {
            i++;                        // too little: take a dearer small item
        } else {
            j--;                        // too much: take a cheaper large item
        }
    }
```

```
Voucher 37000: 7000 + 30000; 12000 + 25000; steps 5 (every pair: 28)
Voucher 12000: 5000 + 7000; steps 7 (every pair: 28)
Voucher 11000: no pair; steps 7 (every pair: 28)
```

Listing 4.12*: on the unsorted prices the walk prints "no pair" three times — its precondition gives no error message. Python's alternative: a set of prices seen so far, $O(n)$ on average. Book: Sections 4.13.4–4.13.6.*

The Problems Side by Side

Table 4.5 The eleven problems of this chapter, the idea each one teaches, its cost, and the trap its tempting wrong solution demonstrates.

Problem	Key idea	Cost	Trap of the tempting wrong solution
4.1 Words	the sort stays, the comparison changes	$O(n^2L)$	plain < compares character codes
4.2 Repeated character	the value is the index: counters	$O(n)$	stopping at the first character seen before
4.3 Money by name	records; sequential search	$O(n)$ per query	binary search on unsorted data
4.4 Importance	two keys in one rule; stability	$O(n^2L)$	two stable sorts in the wrong order
4.5 Addresses	$b + (rC + c)s$	$O(1)$	forgetting s; rows instead of columns
4.6 Counting a key	binary search for boundaries	$O(\log n)$	range $[0, n-1]$ cannot answer n
4.7 Interpolation	estimate the position from the value	$O(\log\log n)$ on average	no range guard: index out of bounds
4.8 Tracing sorts	swaps and shifts equal inversions	$O(n^2)$	inner loop one step too far
4.9 Merging	three indices, then copy the tail	$O(n+m)$	forgetting the leftover tail
4.10 Deleting	read index and write index	$O(n)$	shifting, then skipping the next item
4.11 Voucher pairs	sort, then walk inward	$O(n \log n)$	walking on unsorted data

***Table 4.5**: preconditions, boundaries, and edge examples. Book: Section 4.14.*

Profitina in Practice

A leaderboard with a tie-break is Problem 4.4 in costume: a primary key and a stable secondary order, so the list does not flicker.

"All scans between 1 July and 30 September" is the check-yourself of Problem 4.6: two boundary searches, $O(\log n)$.

The dashboard cockpit gathers many notification kinds, each sorted by time: the merge of Problem 4.9, generalized.

"N new" counters count per channel in one pass: the counting idea of Problem 4.2.

The laundry returns with a queue of orders in Chapter 5, a list of customers in Chapter 7, and a price lookup table in Chapter 15. Book: Section 4.15.

Common Mistakes

Binary search, interpolation, merging or the two-index walk on unsorted data: no error, only wrong answers.

Character codes for dictionary order; misreading a short statement; checking only the first example.

A loop one step too far; a boundary search on $[0, n-1]$; `(lo + hi) / 2` on huge arrays.

Keys of a multi-pass sort in the wrong order; the element size forgotten in an address; dividing before multiplying.

The tail of a merge forgotten; advancing after deleting by shifting; Java strings compared with `==`.

Every tempting wrong solution today passed at least one example of its statement. Book: Section 4.16.

Summary: Five Things to Remember

1. Eight steps: restate, examples by hand, structure and algorithm, trace, cost, code, compare, look back.

2. Estimate before coding: all pairs cost Equation (4.1), binary search at most Equation (4.2).

3. A sort only needs a rule for “which comes first”; change the rule, not the sort (Problems 4.1, 4.4).

4. Counting (4.2), boundaries (4.6), two indices (4.10, 4.11) and the merge (4.9) turn quadratic ideas into linear or logarithmic ones.

5. Preconditions and edges: know what your algorithm assumes, and test the first, the last, the missing and the tied.

Chapter 5 builds the stack and the queue on exactly the array-with-a-size idea of Problem 4.10. Book: Section 4.17.

Next Time

Chapter 5 — Stacks and Queues. Last in first out and first in first out, on arrays: push and pop, enqueue and dequeue.

The Towers of Hanoi, matching brackets, undo and redo, infix to postfix, the circular queue and the deque.

Before then: work Exercises 4.1, 4.3, 4.6 and 4.10 of Chapter 4, by hand first, then in code.

Every exercise has a full solution in the book; try it before you look. Book: Sections 4.18 and 4.19.

Lecture 5 — Stacks and Queues

Course: Data Structure (EF234201)

Book: Irfan Subakti, *Data Structure with C++, Java and Python*, Chapter 5

Lecturer: Irfan Subakti — yifana@gmail.com

Today we take away the array's freedom on purpose: a stack touches only its top, a queue only its two ends, and that restriction makes every operation constant time. Book: Chapter 5, Sections 5.1–5.20.

What This Meeting Covers

1. **The stack** — the Towers of Hanoi, the ADT, an array stack in three languages, growing and linked stacks.
2. **Four jobs a stack does** — matching brackets, undo and redo, infix to postfix and evaluation, the function call stack.
3. **The queue** — the ADT, the shifting and the drifting array queue, and the circular queue that cures both.
4. **Full or empty?** — a count, or one slot left empty; the deque; a preview of the priority queue.
5. **The libraries**, the costs at a glance, Profitina, and the common mistakes.

Every structure is drawn state by state before any code is shown. Book: Section 5.1.

Where We Left Off: From Free Access to Discipline

Chapters 1 to 4 let an algorithm read and write **any** element in any order: the array as a room with an open door.

Pustaka Ceria's **book-return cart**: books go on top and come off the top. Its **new-arrivals line**: first come, first cataloged.

Touching only one known position gives **simplicity** (no search, no shifting: $O(1)$), **correctness by construction**, and **a match with the problem**.

Undoing edits, matching brackets, serving customers and printing documents are last-in-first-out or first-in-first-out by nature. Book: Sections 5.1 and 5.1.1.

Abstract Data Types

> **DEFINITION.** An *abstract data type* (ADT) is a collection of values **together with the operations allowed on it and what those operations do**, with no commitment to how the values are stored. A *data structure* is one concrete way of storing them so that the operations can be carried out.

"Stack" is an ADT: `push`, `pop`, `peek`, `isEmpty`, and `pop` returns the most recently pushed value still present.

An array with a `top` index, a chain of linked nodes and a Python `list` are three data structures that implement it.

Code that uses a stack only through its operations works unchanged with any implementation. Book: Section 5.1.2.

Two Disciplines

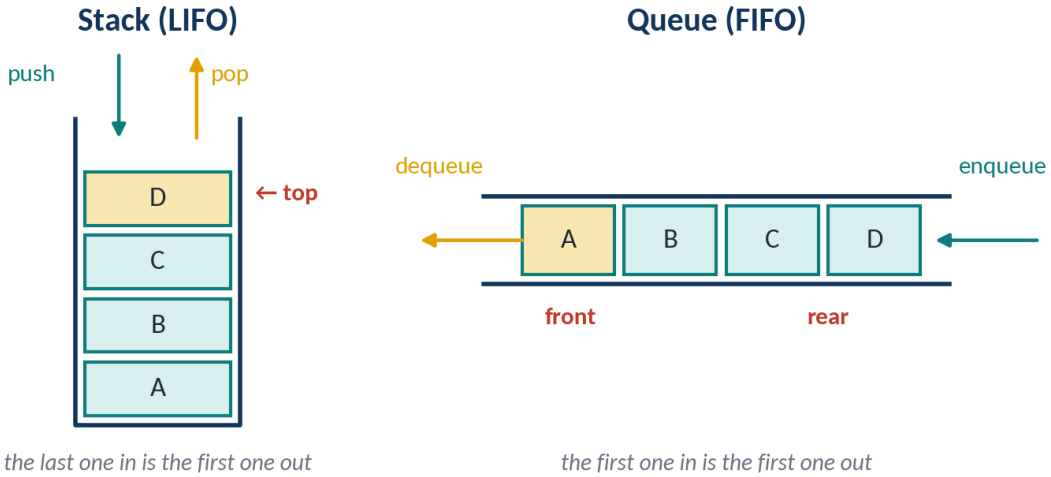


Figure 5.1 The two disciplines of this chapter. A stack adds and removes at the same end (LIFO); a queue adds at the rear and removes at the front (FIFO).

Figure 5.1*: a stack enters and leaves through one opening, last in first out (**LIFO**); a queue enters at the rear and leaves from the front, first in first out (**FIFO**). Book: Section 5.1.2.*

A Story: the Towers of Hanoi

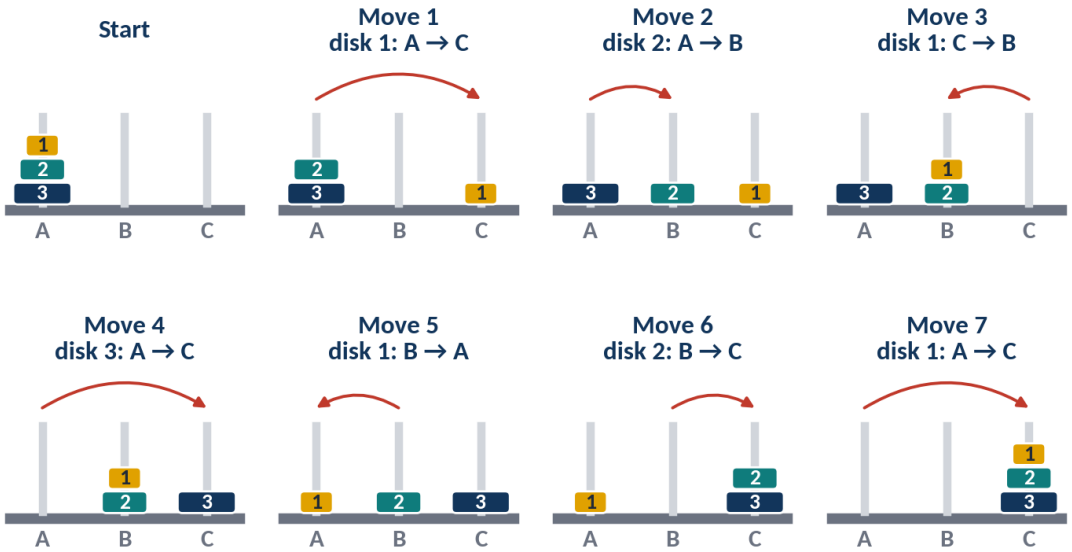


Figure 5.2 The seven moves that transfer three disks from peg A to peg C. The red arrow marks the move that produced each state; at every step only a top disk moves, and it never lands on a smaller disk.

***Figure 5.2**: move one disk at a time, only a top disk, never onto a smaller one. Each peg is a stack. Édouard Lucas sold the puzzle in 1883 with a legend of 64 golden disks. Book: Sections 5.2 and 5.2.1.*

Worked Example 1: Three Disks, and a Count

Moves 1–3 park disks 1 and 2 on B; **move 4** is the only move of disk 3; **moves 5–7** carry the two-disk tower onto it. Solving n disks is solving $n-1$ twice with one move in between, Equation (5.1):

$$M(n) = 2M(n-1) + 1, \qquad M(1) = 1. \tag{5.1}$$

$M(1) = 1, M(2) = 3, M(3) = 7, M(4) = 15, M(5) = 31$: one less than a power of two, Equation (5.2):

$$M(n) = 2^n - 1. \tag{5.2}$$

Check: $2(2^{n-1} - 1) + 1 = 2^n - 1$. *Chapter 10 proves results like this by induction. Book: Sections 5.2.1 and 5.2.2.*

The World Is Safe for a While

64 disks need $2^{64} - 1 = 18{,}446{,}744{,}073{,}709{,}551{,}615$ moves. At one move a second, Equation (5.3):

$$\frac{2^{64} - 1}{60 \cdot 60 \cdot 24 \cdot 365.25} \approx 5.85 \times 10^{11} \text{ years}, \tag{5.3}$$

About 585 billion years: some forty times the present age of the universe.

> **LESSON.** An algorithm whose cost doubles with every extra unit of input becomes hopeless long before the input looks large.

Book: Section 5.2.2.

Hanoi With Three Stacks and No Recursion

Listing 5.1 From `01_hanoi_stacks.cpp`: a move is a push onto the target and a pop from the source, refused if it breaks the size rule.

```cpp
// Move the top disk of `from` onto `to`; refuse a larger disk on a smaller one.
bool moveDisk(Peg& from, Peg& to) {
    if (from.isEmpty()) return false;
    if (!to.isEmpty() && to.peek() < from.peek()) return false;
    to.disks.push_back(from.peek());            // push onto the target ...
    from.disks.pop_back();                      // ... pop from the source
    return true;
}
```

```
start                A: [3 2 1]  B: []       C: []
move 1: disk 1 A -> C  A: [3 2]    B: []       C: [1]
move 2: disk 2 A -> B  A: [3]      B: [2]      C: [1]
move 3: disk 1 C -> B  A: [3]      B: [2 1]    C: []
move 4: disk 3 A -> C  A: []       B: [2 1]    C: [3]
move 5: disk 1 B -> A  A: [1]      B: [2]      C: [3]
move 6: disk 2 B -> C  A: [1]      B: []       C: [3 2]
move 7: disk 1 A -> C  A: []       B: []       C: [3 2 1]
```

Listing 5.1*: on odd moves disk 1 steps along a fixed cycle; on even moves the only legal move not involving disk 1 is made. Nobody indexes the middle of a peg; the loop makes exactly* $2^n - 1$ *moves for* $n = 1..10$*. Book: Section 5.2.3.*

The Stack ADT

> **DEFINITION.** A *stack* is a collection in which insertion and removal both happen at one end, the *top*; the element removed is always the one inserted most recently among those still present: **last in, first out**.

A pile of plates, a stack of books, a tube of shuttlecocks: add on top, take from the top.

In an array $S[0..N-1]$ with an index top, the convention $top = -1$ means empty, so Equation (5.4):

$$\text{size} = top + 1. \tag{5.4}$$

Initializing $top = 0$ and testing $top = -1$ mixes two conventions: a stack that is never empty.

Some books let top count the elements instead; both work, mixing them does not. Book: Sections 5.3 and 5.3.2.

The Operations and Their Costs

Table 5.1 The operations of the stack ADT, their preconditions, and their cost in the array implementation of Section 5.4.

Operation	What it does	Precondition	Cost
`push(x)`	Put x on top of the stack	Stack not full (bounded stacks only)	$O(1)$
`pop()`	Remove the top element and return it	Stack not empty	$O(1)$
`peek()` / `top()`	Return the top element without removing it	Stack not empty	$O(1)$
`isEmpty()`	True when the stack holds no element	—	$O(1)$
`isFull()`	True when a bounded stack has no free slot	—	$O(1)$
`size()`	The number of elements	—	$O(1)$
`clear()`	Remove every element	—	$O(1)$ for an array of plain values

***Table 5.1**: pushing onto a full bounded stack is **overflow**; popping or peeking an empty one is **underflow**. Refuse, throw, or grow — but never read `data[-1]`. Book: Section 5.3.1.*

The Operations in Pseudocode

Pseudocode 5.1 The stack operations on an array $S[0..N-1]$ with index top; $top = -1$ means empty.

```
STACK-EMPTY(S)
 1.  return S.top = -1

STACK-FULL(S)
 2.  return S.top = N - 1

PUSH(S, x)
 3.  if STACK-FULL(S)
 4.      error "overflow"
 5.  S.top ← S.top + 1              // first move top up ...
 6.  S[S.top] ← x                   // ... then write

POP(S)
 7.  if STACK-EMPTY(S)
 8.      error "underflow"
 9.  x ← S[S.top]                   // first read ...
10.  S.top ← S.top - 1              // ... then move top down
11.  return x
```

Pseudocode 5.1: *`PUSH` moves up, then writes; `POP` reads, then moves down: `data[++top] = x`, `return data[top--]`. `PEEK` reads without moving. Book: Section 5.3.2.*

Watching an Array Stack

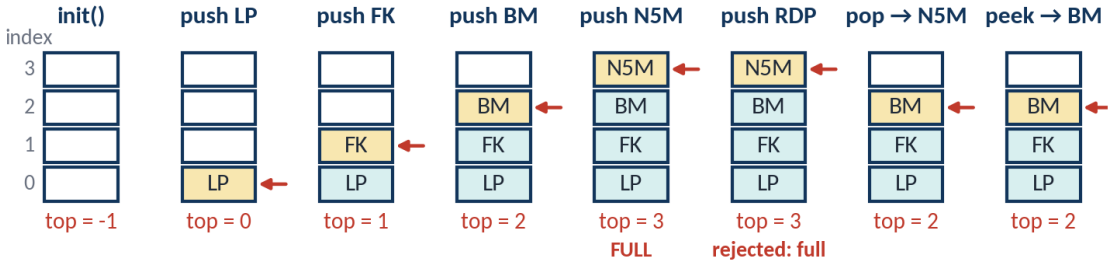


Figure 5.3 An array stack of capacity 4 through eight operations. The amber cell is the top element and the red arrow is the `top` index; after the fourth push the stack is full and the fifth push is rejected.

***Figure 5.3**: four pushes fill the capacity-4 cart; the fifth is refused; `pop` returns N5M and moves the marker — the value stays in slot 3 until overwritten. Removing does not erase. Book: Section 5.4.1.*

The Stack in C++

Listing 5.2 From `02_array_stack.cpp`: `push` reports overflow with `false`; `pop` and `peek` throw on underflow; `clear` is one assignment.

```
    bool push(const T& value) {                         // false = overflow, nothing changed
        if (isFull()) return false;
        data_[++top_] = value;                          // first move top up, then write
        return true;
    }
    T pop() {
        if (isEmpty()) throw underflow_error("pop from an empty stack");
        return data_[top_--];                           // first read, then move top down
    }
    const T& peek() const {
        if (isEmpty()) throw underflow_error("peek at an empty stack");
        return data_[top_];
    }
    void clear() { top_ = -1; }                         // the old values are simply forgotten
```

```
push Negeri 5 Menara -> ok, size 4, top index 3
push Ronggeng Dukuh Paruk -> REJECTED, the cart is full
pop  -> Negeri 5 Menara, size 3, top index 2
```

***Listing 5.2**: every operation is $O(1)$. Copying is deleted, so one array is never freed twice. Java nulls popped slots; Python raises `IndexError`. Book: Sections 5.4.2–5.4.4.*

A Stack That Grows

Listing 5.3 From `03_growing_stack.cpp`: when the array is full, allocate one twice as large and copy.

```
    void push(int value) {
        if (top_ == capacity_ - 1) grow();              // full: double first
        data_[++top_] = value;
    }
    void grow() {
        int* bigger = new int[2 * capacity_];
        for (int i = 0; i <= top_; ++i) bigger[i] = data_[i];   // copy every element
        copies_ += top_ + 1;
        cout << "  grow: capacity " << capacity_ << " -> " << 2 * capacity_
             << ", copied " << top_ + 1 << " element(s)\n";
        delete[] data_;
        data_ = bigger;
        capacity_ *= 2;
    }
```

```
pushing 1..17 onto a stack that starts with capacity 1
  grow: capacity 16 -> 32, copied 16 element(s)
size 17, capacity 32, total copies 31 (less than 2n = 34)
```

***Listing 5.3**: the push that finds 16 elements in 16 slots copies all 16 first. So is `push` still $O(1)$? Not every single push — but on average, yes. Book: Section 5.5.1.*

Amortized $O(1)$

n pushes from capacity 1 double at sizes 1,2,4, … , $2^k < n$, and the copies sum to Equation (5.5):

$$1 + 2 + 4 + \cdots + 2^k = 2^{k+1} - 1 < 2n. \tag{5.5}$$

n writes plus fewer than $2n$ copies: fewer than 3 operations per push on average.

Growing by a fixed 10 slots instead would copy about $n^2/20$: $O(n)$ per push.

The occasional expensive push is paid for by the many cheap ones before it. `std::vector`, `ArrayList`, `ArrayDeque` *and Python's* `list` *all grow by a constant factor. Book: Section 5.5.1.*

A Preview: the Linked Stack

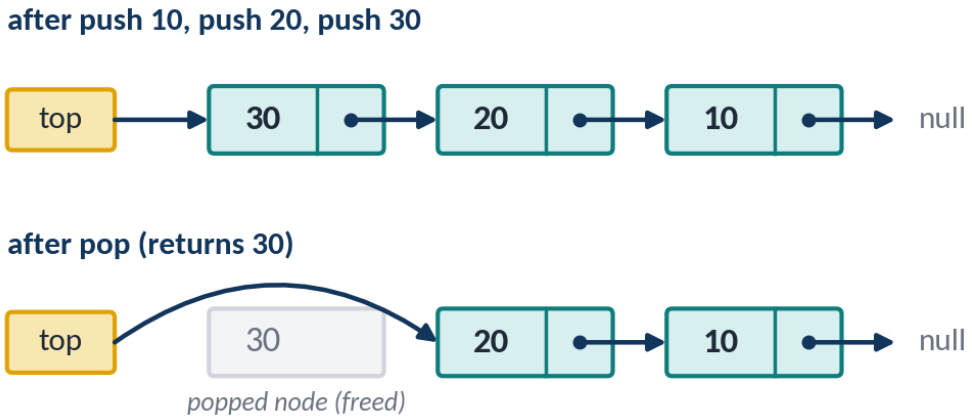


Figure 5.4 A linked stack after pushing 10, 20 and 30 (top), and after one pop (bottom): the top pointer skips the old first node, which is then freed.

***Figure 5.4**: push is `top_ = new Node{v, top_};`; pop unlinks with `top_ = old->next;` before `delete old;`. Never full, $O(1)$ in the worst case, at one allocation and one pointer per element. Chapters 6 and 7 explain the pointers. Book: Section 5.5.2.*

Matching Brackets

Pseudocode 5.2 Checking that brackets nest properly with one stack and one left-to-right scan.

```
BALANCED(text)
 1.  S ← empty stack
 2.  for i ← 0 to length(text) - 1
 3.      c ← text[i]
 4.      if c is one of ( [ {
 5.          PUSH(S, c)
 6.      else if c is one of ) ] }
 7.          if STACK-EMPTY(S)
 8.              return "error at i: c has no opening partner"
 9.          if PEEK(S) is not the partner of c
10.              return "error at i: c cannot close PEEK(S)"
11.          POP(S)
12.      // any other character is ignored
13.  if not STACK-EMPTY(S)
14.      return "error at end: brackets never closed"
15.  return "balanced"
```

***Pseudocode 5.2**: counting is not enough — `([)]` has two of each and is wrong. Every closer must match the most recent unmatched opener: the top of a stack. Three ways to fail: a closer with nothing open (line 8), a mismatch (line 10), an opener never closed (line 14). Book: Sections 5.6 and 5.6.1.*

Brackets, Traced

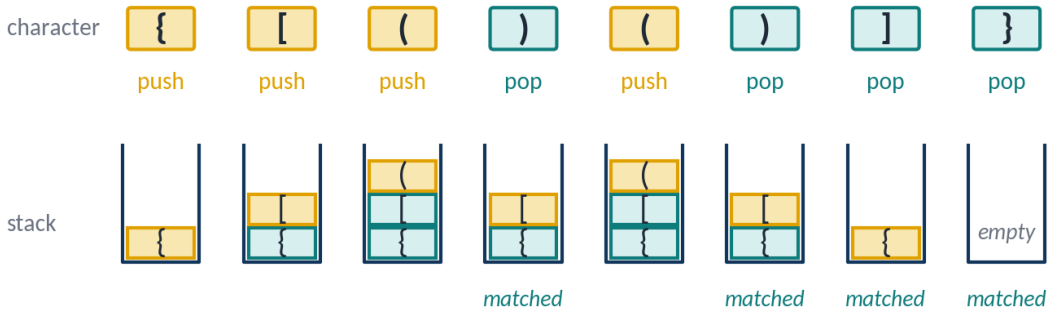


Figure 5.5 Checking `{[()()]}`: each opener is pushed, each closer pops its partner from the top, and the stack is empty at the end, so the text is balanced.

***Figure 5.5**: `{[()()]}` — each opener pushed, each closer pops its partner, and the stack is empty at the end: balanced. Book: Section 5.6.1.*

Brackets in C++

Listing 5.4 From `05_brackets.cpp`: the closing-bracket cases and the check at the end, with the position of the first error.

```
        } else if (isClose(c)) {
            if (open.empty())
                return "error at " + to_string(i) + ": '" + c + "' has no opening partner";
            if (open.top() != partner(c))
                return "error at " + to_string(i) + ": '" + c + "' cannot close '" + open.top() + "'";
            open.pop();
            if (trace) cout << "  " << c << "  pop, matches   stack: " << contents(open) << "\n";
        }                                                  // every other character is ignored
    }
    if (!open.empty())
        return "error at end: " + to_string(open.size()) + " bracket(s) never closed, top is '"
               + open.top() + "'";
    return "balanced";
```

```
"{[()()]}"                  balanced
"([)]"                      error at 2: ')' cannot close '['
"((a+b)*c"                  error at end: 1 bracket(s) never closed, top is '('
"a+b)*(c"                   error at 3: ')' has no opening partner
"if (a[i] == b[j]) { x(); }"  balanced
""                          balanced
```

***Listing 5.4**, with `std::stack<char>`: each failure mode appears once. One pass, $O(1)$ stack work per character: $O(n)$ time and, for n openers, $O(n)$ space. Book: Section 5.6.2.*

Undo and Redo: Two Stacks

Edit: push the current state onto *undo*, make the edit, **clear *redo***.

Undo: push the current state onto *redo*, pop *undo*. **Redo:** push onto *undo*, pop *redo*.

Listing 5.5 From `06_undo_redo.cpp`: undo and redo move one saved state between the two stacks.

```
    void undo() {
        if (undo_.empty()) { report("undo (nothing to undo)"); return; }
        redo_.push(text_);
        text_ = undo_.top(); undo_.pop();
        report("undo");
    }
    void redo() {
        if (redo_.empty()) { report("redo (nothing to redo)"); return; }
        undo_.push(text_);
        text_ = redo_.top(); redo_.pop();
        report("redo");
    }
```

Listing 5.5 *stores whole copies of the text (a memento); real editors store operations, with the same two-stack discipline. The browser's Back and Forward buttons are the same two stacks. Book: Section 5.7.*

Undo and Redo, Drawn

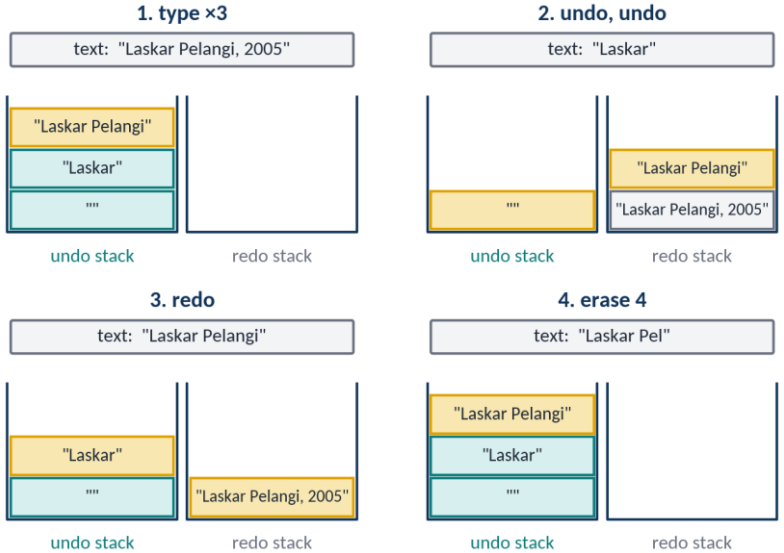


Figure 5.6 The undo and redo stacks at four moments of the run. A new edit (panel 4) pushes onto the undo stack and empties the redo stack.

***Figure 5.6**: after `erase 4`, a new edit, the redo stack is emptied — so the next `redo` reports "nothing to redo". Keeping it would resurrect a state from an abandoned history. Book: Section 5.7.*

Three Ways to Write the Same Sum

Table 5.2 The same expressions in infix, prefix and postfix notation. Postfix needs no parentheses, and its operators appear in exactly the order they are applied.

Infix	Prefix (Polish)	Postfix (reverse Polish)	Value
`3 + 2 * 5`	`+ 3 * 2 5`	`3 2 5 * +`	13
`(3 + 2) * 5`	`* + 3 2 5`	`3 2 + 5 *`	25
`a + b * c - d`	`- + a * b c d`	`a b c * + d -`	9 when $a = 2, b = 3, c = 4, d = 5$
`8 - 3 - 2`	`- - 8 3 2`	`8 3 - 2 -`	3
`2 ^ 3 ^ 2`	`^ 2 ^ 3 2`	`2 3 2 ^ ^`	512

***Table 5.2**: in postfix the operators appear in exactly the order they are applied, so no parentheses are needed — which is why a machine with a stack can evaluate it in one pass (Hamblin, Bauer, Dijkstra, late 1950s). Book: Section 5.8.1.*

Precedence and Associativity

Table 5.3 Operator precedence and associativity used by the calculator.

Operator	Meaning	Precedence	Associativity
`^`	power	3 (highest)	right: `2^3^2` means `2^(3^2)`
`*`, `/`	multiply, divide	2	left: `100/10/5` means `(100/10)/5`
`+`, `-`	add, subtract	1 (lowest)	left: `8-3-2` means `(8-3)-2`

Precedence decides between different operators: `*` beats `+`.

Associativity decides between equal ones: `8-3-2` is $(8-3)-2=3$, not 7; `2^3^2` is $2^9=512$, not $8^2=64$.

In the shunting yard, operands pass straight through; `(` is a barrier; `)` pops back to it; an operator first pops what must be applied before it, then waits on the stack.

Table 5.3 *gives the rules of ordinary school arithmetic, with* `^` *for power. Book: Sections 5.8.2 and 5.8.3.*

Dijkstra's Shunting-Yard Algorithm

Pseudocode 5.3 Dijkstra's shunting-yard algorithm: infix to postfix with one operator stack.

```
INFIX-TO-POSTFIX(tokens)
 1.  S ← empty stack;  out ← empty list
 2.  for each token t, from left to right
 3.      if t is an operand (number or variable)
 4.          append t to out
 5.      else if t = "("
 6.          PUSH(S, t)
 7.      else if t = ")"
 8.          while PEEK(S) ≠ "("
 9.              append POP(S) to out
10.          POP(S)                          // discard the "("; error if S ran empty
11.      else                                // t is an operator
12.          while S not empty and PEEK(S) ≠ "(" and
                  ( prec(PEEK(S)) > prec(t) or
                    ( prec(PEEK(S)) = prec(t) and t is left-associative ) )
13.              append POP(S) to out
14.          PUSH(S, t)
15.  while S not empty
16.      append POP(S) to out                // error if a "(" is left
17.  return out
```

***Pseudocode 5.3**, Dijkstra 1961: a railway shunting yard. Book: Section 5.8.3.*

Worked Example 2: `3+2*5`

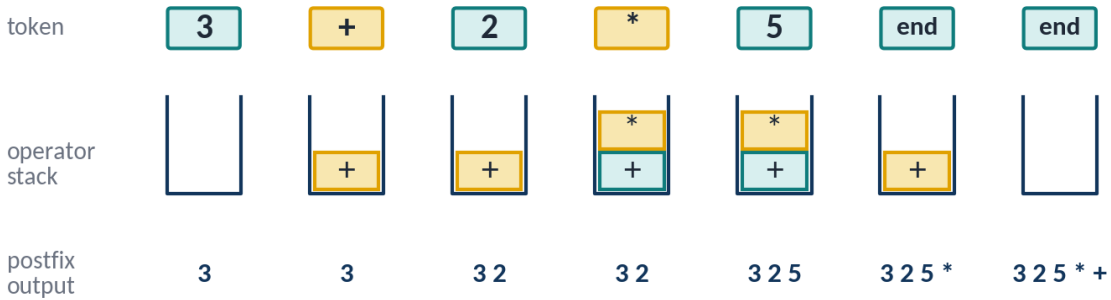


Figure 5.7 Converting `3+2*5` to postfix. The `*` is pushed on top of `+` because it binds more tightly; at the end the stack is emptied from the top, giving `3 2 5 * +`.

***Figure 5.7**: when `*` meets `+`, $\text{prec}(+) < \text{prec}(*)$, so nothing is popped and `*` waits on top. That one comparison makes multiplication happen first. Result: `3 2 5 * +`. Book: Section 5.8.4.*

Worked Example 3: `a+b*c-d`

Table 5.4 Converting `a+b*c-d` to postfix, one token per row.

Token	Action	Stack (bottom → top)	Output so far
`a`	output operand	(empty)	`a`
`+`	push `+`	`+`	`a`
`b`	output operand	`+`	`a b`
`*`	prec(+) < prec(∗): push `*`	`+ *`	`a b`
`c`	output operand	`+ *`	`a b c`
`-`	pop `*` (higher), pop `+` (equal, left-assoc.), push `-`	`-`	`a b c * +`
`d`	output operand	`-`	`a b c * + d`
end	pop `-`	(empty)	`a b c * + d -`

***Table 5.4**: when `-` arrives, `*` is popped (higher) and `+` too (equal, left-associative). `a b c * + d -` is exactly $(a + (b \times c)) - d$. For the right-associative `^`, the equal case does **not** pop. Book: Section 5.8.5.*

Evaluating Postfix

Pseudocode 5.4 Evaluating a postfix expression with one operand stack.

```
EVALUATE-POSTFIX(tokens)
 1.  S ← empty stack
 2.  for each token t, from left to right
 3.      if t is an operand
 4.          PUSH(S, value of t)
 5.      else                                  // t is a binary operator
 6.          if S holds fewer than 2 values
 7.              error "operator needs two operands"
 8.          y ← POP(S)                        // the FIRST pop is the RIGHT operand
 9.          x ← POP(S)
10.          PUSH(S, x t y)
11.  if S does not hold exactly one value
12.      error "malformed expression"
13.  return POP(S)
```

THE OPERAND TRAP. For `8 3 -` the first pop returns 3, the second 8: compute $x - y = 8 - 3 = 5$. Addition and multiplication hide the bug; subtraction and division expose it.

Pseudocode 5.4*: push numbers; an operator pops its two operands, applies itself, and pushes the result. Exactly one number must remain. Book: Section 5.9.*

Postfix, Evaluated

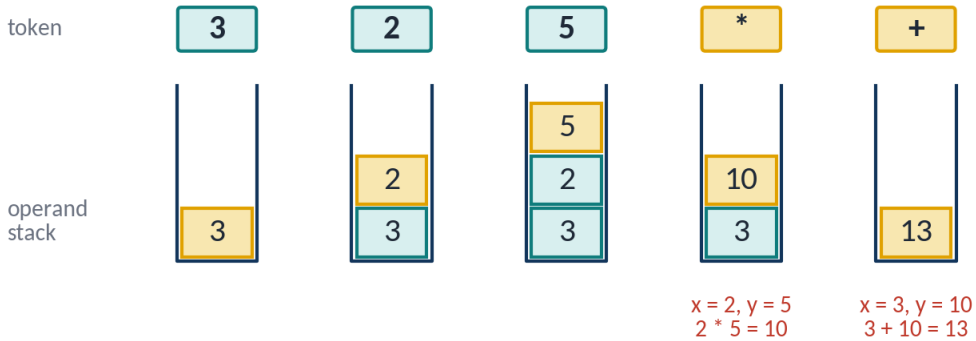


Figure 5.8 Evaluating `3 2 5 * +`. Operands are pushed; each operator pops y (the top) and then x, and pushes x op y. The single value left, 13, is the result.

***Figure 5.8**: `3 2 5 * +` — `*` pops 5 and 2 and pushes 10; `+` pops 10 and 3 and pushes 13. Book: Section 5.9.*

The Calculator's Evaluator

Listing 5.6 From `07_calculator.cpp`: the operand stack; the first pop is the right operand.

```
    for (const string& t : postfix) {
        string action;
        if (isOperator(t)) {
            if (st.size() < 2) throw runtime_error("operator " + t + " needs two operands");
            long long y = st.back(); st.pop_back();     // the FIRST pop is the right operand
            long long x = st.back(); st.pop_back();
            st.push_back(apply(x, t, y));
            action = to_string(x) + " " + t + " " + to_string(y) + " = " + to_string(st.back());
        } else {
            long long v = isdigit(static_cast<unsigned char>(t[0])) ? stoll(t) : vars.at(t[0]);
            st.push_back(v);
            action = "push " + to_string(v);
        }
        if (trace) row(t, action, show(), "");
    }
    if (st.size() != 1) throw runtime_error("malformed expression");
    return st.back();
```

Listing 5.6 *after* `toPostfix` *— conversion and evaluation are each $O(n)$ for n tokens. Python needs one extra line: its* `//` *rounds toward minus infinity, while C++ and Java truncate toward zero. Book: Section 5.9.1.*

The Calculator at Work

```
(7+3)*(10-4)/5  postfix 7 3 + 10 4 - * 5 /    = 12
2^3^2           postfix 2 3 2 ^ ^             = 512
(2^3)^2         postfix 2 3 ^ 2 ^             = 64
100/10/5        postfix 100 10 / 5 /          = 2
8-3-2           postfix 8 3 - 2 -             = 3
(1+2            error: unmatched '('
1+2)            error: unmatched ')'
1+              postfix 1 +                   error: operator + needs two operands
7/(3-3)         postfix 7 3 3 - /             error: division by zero
```

Parentheses and multi-digit numbers: $10 \cdot 6/5 = 12$. Right associativity: 512 against 64.

Left associativity with the operands in the right order: `100/10/5` is 2, `8-3-2` is 3.

Unmatched parentheses are caught in conversion, a missing operand and division by zero in evaluation.

The traces for `3+2*5` and `a+b*c-d` match the figures token for token: 13 and 9. Book: Section 5.9.1.

The Function Call Stack

Each call pushes a **stack frame**: return address, parameters, local variables, saved registers. Each return pops it.

A local variable disappears when its function returns: never return a pointer or reference to one.

```
push  main
push  main | lateFee(7)
push  main | lateFee(7) | dailyRate(reference)
pop   main | lateFee(7)
push  main | lateFee(7) | capFee(35000)
pop   main | lateFee(7)
pop   main
        late fee = Rp25000
```

*`08_call_stack.cpp` pushes a name on entry and pops it in a destructor on every return. Recursion is frames of the same function on top of each other; a runaway recursion is a **stack overflow** (1 MB by default on Windows, typically 8 MB on Linux). Book: Section 5.10.*

The Call Stack, Drawn

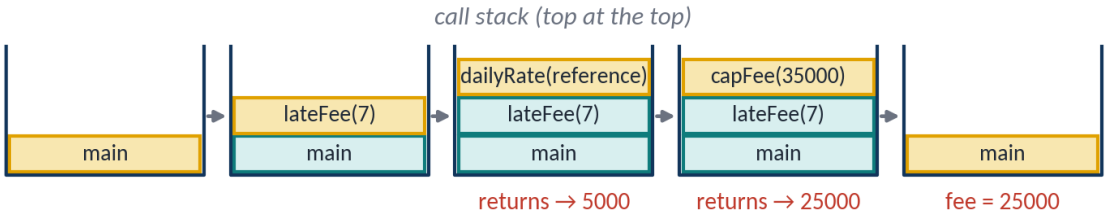


Figure 5.9 The call stack during `lateFee(7, "reference")`: each call pushes a frame on top, and each return pops it, handing its value to the frame below.

***Figure 5.9**: `dailyRate` is called in the middle of computing the fee, returns 5000 and is popped before `capFee` is even called; the fee $7 \times 5000 = 35{,}000$ is capped at Rp25,000. Book: Section 5.10.2.*

The Queue ADT

> **DEFINITION.** A *queue* inserts at the *rear* and removes at the *front*; the element removed is always the one that has waited longest: **first in, first out**.

Print jobs, round-robin process scheduling, router buffers, keyboard buffers, breadth-first search (Chapter 13).

Table 5.5 The operations of the queue ADT and their cost in the circular-array implementation of Section 5.13.

Operation	What it does	Precondition	Cost (circular array)
`enqueue(x)`	Add x at the rear	Queue not full (bounded queues only)	$O(1)$
`dequeue()`	Remove the front element and return it	Queue not empty	$O(1)$
`peek()` / `front()`	Return the front element without removing it	Queue not empty	$O(1)$
`isEmpty()`	True when the queue holds no element	—	$O(1)$
`isFull()`	True when a bounded queue has no free slot	—	$O(1)$
`size()`	The number of elements	—	$O(1)$

***Table 5.5**: `enqueue` is also add, offer, push or put; `dequeue` is also remove, poll, pop or get. Book: Sections 5.11 and 5.11.1.*

A Queue Through Seven Operations

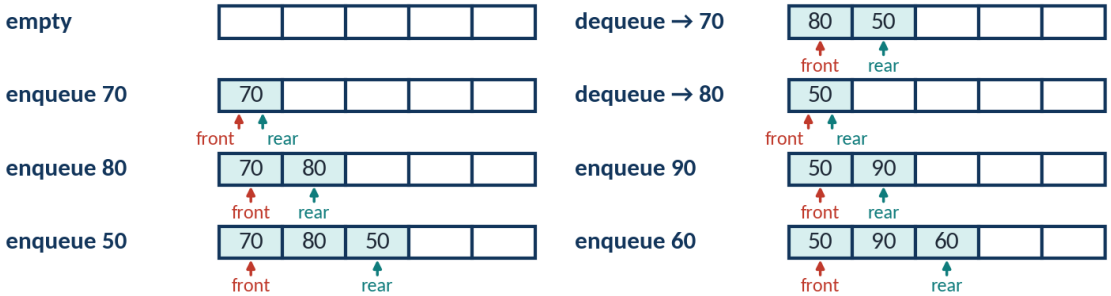


Figure 5.10 A queue through seven operations, drawn with the front on the left. Elements leave in exactly the order they arrived: 70, then 80.

***Figure 5.10**: 50 is the latest arrival at the first dequeue, yet 70 leaves, then 80. The figure hides the storage; a queue changes at **both** ends, which makes arrays harder. Book: Section 5.11.1.*

First Attempt: Shift on Every Dequeue

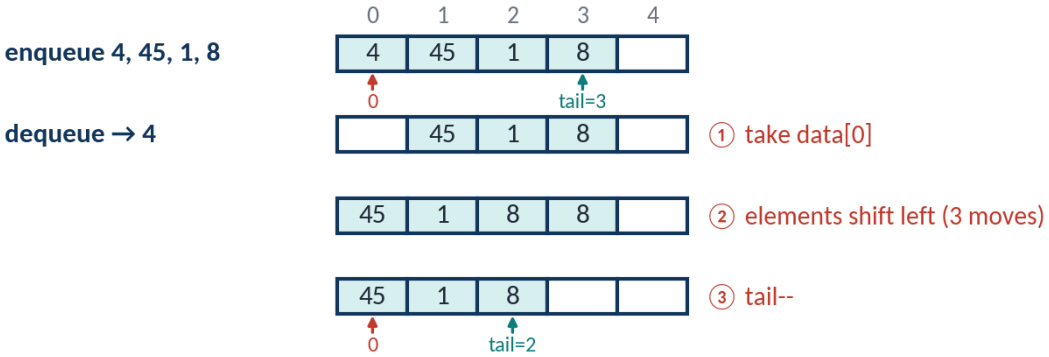


Figure 5.11 The shifting queue: dequeue takes slot 0 (step 1), shifts every other element one place left (step 2), and decrements `tail` (step 3). A queue of n elements costs $n - 1$ moves per dequeue.

***Figure 5.11**: correct and never wastes a slot, but every dequeue moves $n - 1$ elements: $O(n)$. Forgetting `tail--` duplicates the last element (`45 1 8 8`). Book: Section 5.12.1.*

Second Attempt: the Drifting Queue

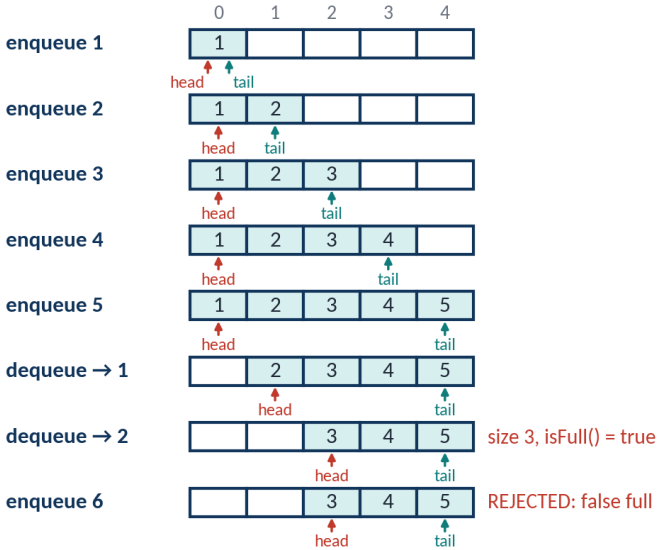


Figure 5.12 The drifting queue. Both indices only move right. After two dequeues, slots 0 and 1 are free, but `tail` has reached the last slot, so `isFull()` reports true and the next enqueue is refused: a *false full*.

***Figure 5.12**: both indices only move right, so every slot left of `head` is lost. `09_linear_queue.cpp` ends with "size 3 of 5, but isFull() says true": a **false full**. Book: Section 5.12.2.*

The Circular Queue

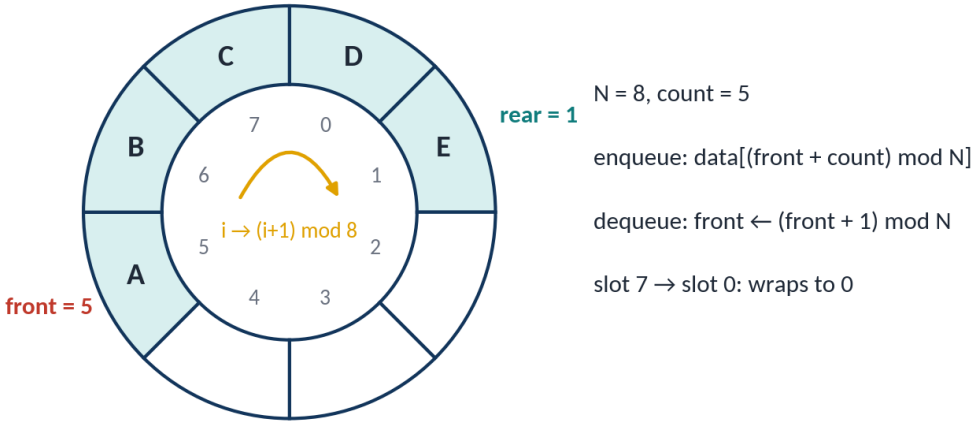


Figure 5.13 A circular queue of capacity $N = 8$ holding A to E. The queue starts at `front` = 5 and runs clockwise through slots 6, 7, 0 and 1: after slot 7 it wraps to slot 0.

$$i \leftarrow (i + 1) \bmod N. \tag{5.6}$$

***Figure 5.13**: bend the array into a ring. Equation (5.6) takes $7 \rightarrow (7 + 1) \bmod 8 = 0$: the index never leaves the array, and no `if` is needed. Book: Sections 5.13 and 5.13.1.*

Front Plus a Count

$$\text{free} = (\text{front} + \text{count}) \bmod N, \tag{5.7}$$

Pseudocode 5.5 `ENQUEUE` and `DEQUEUE` on a circular array of N slots with a `front` index and a `count`.

```
ENQUEUE(Q, x)
 1.  if Q.count = N
 2.      error "overflow"
 3.  Q[(Q.front + Q.count) mod N] ← x     // first free slot, wrapped
 4.  Q.count ← Q.count + 1

DEQUEUE(Q)
 5.  if Q.count = 0
 6.      error "underflow"
 7.  x ← Q[Q.front]
 8.  Q.front ← (Q.front + 1) mod N        // advance and wrap
 9.  Q.count ← Q.count - 1
10.  return x
```

*Equation (5.7): the first free slot is computed, not stored; the last element is at $(front + count - 1) \bmod N$. **Pseudocode 5.5**: empty when $count = 0$, full when $count = N$. Book: Section 5.13.2.*

Worked Example 4: the New-Arrivals Line

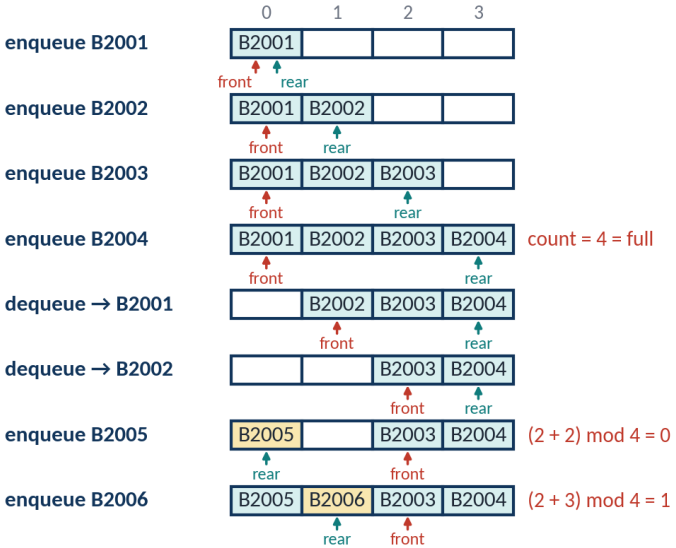


Figure 5.14 The circular queue of the new-arrivals line with capacity 4. The last two enqueues wrap into slots 0 and 1, so the rear is now to the left of the front, and the queue is still in perfect FIFO order.

Figure 5.14*: after two dequeues* `front` *is 2; B2005 goes to* $(2 + 2)\ mod\ 4 = 0$ *and B2006 to* $(2 + 3)\ mod\ 4 = 1$*. Physically B2005 B2006 B2003 B2004; as a queue B2003 B2004 B2005 B2006. Book: Section 5.13.2.*

The Circular Queue in C++

Listing 5.7 From `10_circular_queue.cpp`: enqueue at the wrapped slot after the rear; dequeue advances `front` and wraps.

```
    int  rear()    const { return (front_ + count_ - 1 + capacity_) % capacity_; }

    bool enqueue(const T& value) {
        if (isFull()) return false;
        data_[(front_ + count_) % capacity_] = value;   // the slot just after the rear, wrapped
        ++count_;
        return true;
    }
    T dequeue() {
        if (isEmpty()) throw underflow_error("dequeue from an empty queue");
        T value = data_[front_];
        front_ = (front_ + 1) % capacity_;               // advance and wrap
        --count_;
        return value;
    }
```

NEGATIVE REMAINDERS. In C++ and Java `-1 % 4` is `-1`, not 3: `rear()` adds `capacity_` before taking `%`. Python's `%` is always non-negative.

Listing 5.7*: the run matches the figure row for row, and prints "cataloging order: B2003 B2004 B2005 B2006". Book: Section 5.13.3.*

The Circular Queue in Python

Listing 5.8 From `10_circular_queue.py`: the same two operations; the dequeued slot is set to `None`.

```
    def enqueue(self, value):
        if self.is_full():
            return False
        self._data[(self._front + self._count) % len(self._data)] = value   # just after the rear, wrapped
        self._count += 1
        return True

    def dequeue(self):
        if self.is_empty():
            raise IndexError("dequeue from an empty queue")
        value = self._data[self._front]
        self._data[self._front] = None
        self._front = (self._front + 1) % len(self._data)                    # advance and wrap
        self._count -= 1
        return value
```

```
enqueue B2005             [ B2005 ..... B2003 B2004 ]  front=2 rear=0 count=3
enqueue B2006             [ B2005 B2006 B2003 B2004 ]  front=2 rear=1 count=4
peek -> B2003
cataloging order: B2003 B2004 B2005 B2006
```

***Listing 5.8**: C++, Java and Python print the same fourteen lines. Book: Section 5.13.3.*

Second Design: Leave One Slot Empty

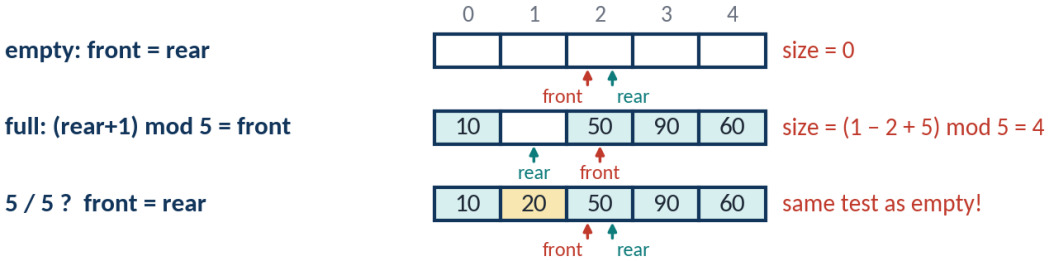


Figure 5.15 Without a count, `front = rear` would mean both "empty" (top) and "all five slots used" (bottom). Keeping one slot always free (middle) makes the full state distinct: the queue is full when advancing `rear` would reach `front`.

***Figure 5.15**: with only `front` and `rear` (next free slot), an empty queue and a full one both have `front = rear`. Never fill the last slot, and the two states differ. Book: Section 5.13.4.*

Count or Empty Slot?

$$\text{size} = (\text{rear} - \text{front} + N) \bmod N. \tag{5.8}$$

$$\text{full} \Leftrightarrow (\text{rear} + 1) \bmod N = \text{front}, \qquad \text{empty} \Leftrightarrow \text{rear} = \text{front}. \tag{5.9}$$

Table 5.6 The two ways to distinguish a full circular queue from an empty one.

	Count design (Section 5.13.2)	One-empty-slot design (Section 5.13.4)
Stored fields	`front`, `count`	`front`, `rear`
Meaning of rear	computed: (front + count − 1) mod N is the last element	stored: `rear` is the next free slot
Empty test	count $= 0$	rear = front
Full test	count $= N$	(rear + 1) mod N = front
Size	`count`	(rear − front + N) mod N
Usable slots	N	$N - 1$
Typical use	teaching code, most library queues	lock-free buffers where one thread writes `rear` and another writes `front`

*Equation (5.8) and Equation (5.9): an array of N slots holds at most $N - 1$. **Table 5.6**: when one thread only enqueues and another only dequeues, each writes only its own index — hence ring buffers in audio drivers and network cards. Book: Section 5.13.4.*

The Deque

> **DEFINITION.** A *deque* (“deck”) supports `pushFront`, `pushBack`, `popFront`, `popBack`, `peekFront` and `peekBack`. With only the back it is a stack; with back in and front out, a queue.

`pushFront` steps `front` **backward**, adding N first to keep the remainder non-negative, Equation (5.10):

$$\text{front} \leftarrow (\text{front} - 1 + N) \bmod N. \tag{5.10}$$

`popBack` only decrements `count`, since the back position is computed from it. Book: Section 5.14.

A Circular-Array Deque

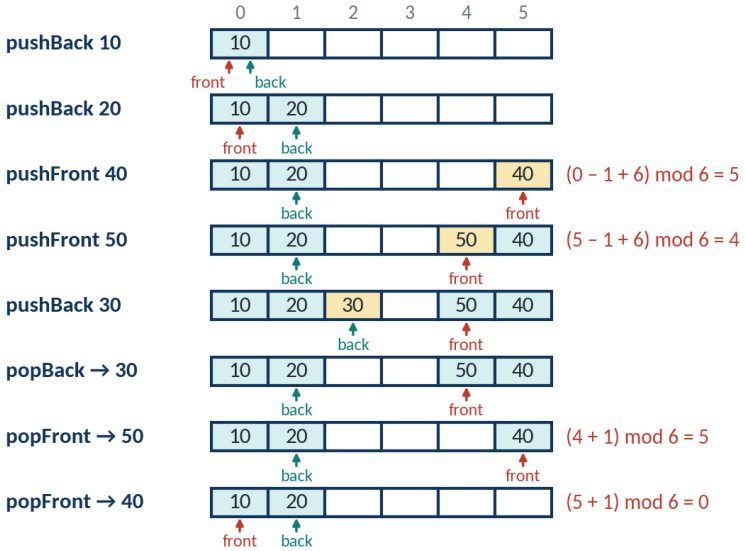


Figure 5.16 A circular-array deque of capacity 6. `pushFront` moves `front` backward with wrap-around, from 0 to 5 and then to 4; `popFront` moves it forward again.

***Figure 5.16**: the first `pushFront` gives $(0 - 1 + 6) \bmod 6 = 5$, so 40 lands in the last slot, logically in front of 10. Book: Section 5.14.*

The Deque in C++, and a Palindrome

Listing 5.9 From `12_array_deque.cpp`: `pushFront` steps back with wrap-around; `popBack` reads the old back slot.

```
    void pushFront(const T& v) {
        if (isFull()) throw overflow_error("deque is full");
        front_ = (front_ - 1 + cap_) % cap_;           // step back one slot, wrapping to the end
        data_[front_] = v;
        ++count_;
    }
    T popBack() {
        if (isEmpty()) throw underflow_error("deque is empty");
        --count_;
        return data_[(front_ + count_) % cap_];         // the old back slot
    }
```

```
"kasur rusak"                    palindrome
"Taat"                           palindrome
"racecar"                        palindrome
"stack"                          not a palindrome
"A man, a plan, a canal: Panama"  palindrome
```

Listing 5.9*: the palindrome test pops both ends and compares, $O(n)$. Deques also drive work-stealing schedulers and the $O(n)$ sliding-window maximum (Exercise 5.14). Book: Section 5.14.*

Priority Queues: a Preview

Table 5.7 Three ways to implement a priority queue. The heap of Chapter 14 is the only one that makes both operations fast.

Implementation	`insert`	`extractMax`	Idea
Unsorted array	$O(1)$	$O(n)$	append at the end; scan for the maximum
Sorted array	$O(n)$	$O(1)$	shift to keep order; the maximum is at the end
Binary heap (Chapter 14)	$O(\lg n)$	$O(\lg n)$	a tree hidden in an array

***Table 5.7**: an emergency room serves the highest priority first, whatever the arrival time. An unsorted or sorted array makes one of the two operations linear; the binary heap of Chapter 14 makes both $O(lgn)$ — about 20 steps for a million elements. Book: Section 5.15.*

Stacks and Queues in the Standard Libraries

Table 5.8 The same operations in the three standard libraries.

Operation	C++	Java	Python
stack push / pop / top	`s.push(x)` / `s.pop()` / `s.top()`	`d.push(x)` / `d.pop()` / `d.peek()`	`l.append(x)` / `l.pop()` / `l[-1]`
queue enqueue / dequeue / front	`q.push(x)` / `q.pop()` / `q.front()`	`d.offer(x)` / `d.poll()` / `d.peek()`	`d.append(x)` / `d.popleft()` / `d[0]`
deque both ends	`push_front`, `push_back`, `pop_front`, `pop_back`	`addFirst`, `addLast`, `pollFirst`, `pollLast`	`appendleft`, `append`, `popleft`, `pop`
priority queue	`std::priority_queue` (max first)	`PriorityQueue` (min first)	`heapq` on a list (min first)
empty test	`s.empty()`	`d.isEmpty()`	`not l` or `len(l) == 0`
on empty pop	undefined behavior	`pop()` throws, `poll()` returns `null`	`IndexError`
thread-safe queue	none in the standard library	`java.util.concurrent.ArrayBlockingQueue`	`queue.Queue`

***Table 5.8**: in C++ `pop()` returns `void` and popping an empty adaptor is undefined behavior; `std::priority_queue` is a max-heap. In Java use `ArrayDeque`, not the legacy `Stack`. In Python never use `list.pop(0)` as a queue. Book: Section 5.16.*

The Python Tour

Listing 5.10 From `13_library.py`: a `list` as a stack and a `collections.deque` as a queue.

```
    s = []                                  # a list used as a stack: append/pop at the end
    for v in (1, 2, 3):
        s.append(v)
    out = []
    top = s[-1]
    while s:
        out.append(s.pop())
    print(f"stack  : push 1 2 3 -> top {top}, pop order: " + " ".join(map(str, out)))

    q = deque()                             # a deque used as a queue: append right, popleft left
    for v in (1, 2, 3):
        q.append(v)
    front, back = q[0], q[-1]
    out = []
    while q:
        out.append(q.popleft())
    print(f"queue  : push 1 2 3 -> front {front}, back {back}, pop order: " + " ".join(map(str, out)))
```

```
stack  : push 1 2 3 -> top 3, pop order: 3 2 1
queue  : push 1 2 3 -> front 1, back 3, pop order: 1 2 3
```

***Listing 5.10**: `append` and `pop()` at the end of a list are amortized $O(1)$; `deque.popleft()` is $O(1)$. `heapq` is a min-heap, so the largest-first line pushes —v. Book: Section 5.16.3.*

Costs at a Glance

Table 5.9 The cost of every structure in this chapter.

Structure	Add	Remove	Peek	Space for n items	Fixed capacity?
Array stack (Section 5.4)	$O(1)$	$O(1)$	$O(1)$	N slots	yes
Doubling array stack (Section 5.5.1)	$O(1)$ amortized, $O(n)$ worst	$O(1)$	$O(1)$	at most $2n$ slots	no
Linked stack (Section 5.5.2)	$O(1)$	$O(1)$	$O(1)$	n nodes + n pointers	no
Shifting linear queue (Section 5.12.1)	$O(1)$	$O(n)$	$O(1)$	N slots	yes
Drifting linear queue (Section 5.12.2)	$O(1)$	$O(1)$	$O(1)$	slots lost for good	yes, and shrinking
Circular queue (Section 5.13)	$O(1)$	$O(1)$	$O(1)$	N (or $N-1$) slots	yes
Circular deque (Section 5.14)	$O(1)$ both ends	$O(1)$ both ends	$O(1)$	N slots	yes
Priority queue, binary heap (Chapter 14)	$O(\lg n)$	$O(\lg n)$	$O(1)$	n slots	no

***Table 5.9**: most recent first — stack; oldest first — queue, as a circular buffer, never a drifting one; both ends — deque; most important first — priority queue. In production, use the library's version. Book: Section 5.17.*

Profitina in Practice

Structured data must be balanced: one missing `}` in a JSON-LD block hides it from answer engines — the check of **Pseudocode 5.2**.

A crawl is a queue: fetching discovered pages level by level is breadth-first search.

Slow work goes into a bounded job queue: FIFO is fair, and a full queue says so rather than accepting work it cannot finish.

Scheduled re-scans are a priority queue keyed by due time; **a live feed needs bounded ring buffers**; **an optimistic post is a one-level undo**.

The feedback pipeline — submit, curate, dataset, train and measure, release — is a queue of queues. Book: Section 5.18.

Common Mistakes

`data[top++] = x` with $top = -1$, or mixing the two `top` conventions; popping without checking.

Swapping the operands in postfix evaluation; treating `^` as left-associative; forgetting the parenthesis barrier.

Forgetting `tail--`; shipping a drifting queue; negative remainders in C++ and Java.

`front == rear` **for both full and empty**; `list.pop(0)` or `java.util.Stack`; keeping the redo stack after a new edit.

Returning the address of a local; copying a class that owns an array.

Fourteen mistakes, each with its fix, are listed in the book. Book: Section 5.19.

Summary: Five Things to Remember

1. A stack is LIFO: `push`, `pop`, `peek` in $O(1)$ with $top = -1$ meaning empty (Equation (5.4)); a doubling array is amortized $O(1)$ (Equation (5.5)).

2. Hanoi needs $2^n - 1$ moves (Equation (5.2)) and can be solved with three stacks and a loop.

3. Stacks match brackets, undo and redo, convert infix to postfix and evaluate it, and run every function call.

4. A queue is FIFO; the circular queue wraps with Equation (5.6) and tells full from empty by a count or an empty slot (Equation (5.9)).

5. A deque works at both ends; a priority queue serves the most important first; the libraries provide them all.

Book: Section 5.20.

Next Time

Chapter 6 — Pointers, References, and Functions. The pointer story begun in Chapter 1, completed: `new`, `delete`, references, and how functions receive their arguments.

The linked stack of today is the first structure built from pointers; Chapter 7 builds whole lists from them.

Before then: work Exercises 5.2, 5.5, 5.9 and 5.14.

Exercise 5.15 designs Profitina Laundry's order counter with the structures of today. Book: Sections 5.21 and 5.22.

Lecture 6 — Pointers, References, and Functions

Course: Data Structure (EF234201)

Book: Irfan Subakti, *Data Structure with C++, Java and Python*, Chapter 6

Lecturer: Irfan Subakti — yifana@gmail.com

Today we finish the pointer story that Chapter 1 began: addresses, dynamic memory, the bugs that follow, the three ways a function receives an argument, and what Java and Python do instead. Book: Chapter 6, Sections 6.1–6.18.

What This Meeting Covers

1. **What a pointer is** — `&`, `*`, the box-and-arrow drawing, assignment, comparison and arithmetic.
2. **Pointers and arrays** — decay, `a[i] ≡ *(a + i)`, strings and their terminator, arrays as parameters, the arrow operator and a first chain of nodes.
3. **Dynamic memory** — `new` and `delete`, the doubling array, and the five classic bugs caught by AddressSanitizer and Valgrind.
4. **`nullptr`, references, the three ways to pass an argument, `const`, returning safely.**
5. **Smart pointers, Java references and Python names**, Profitina Laundry.

Everything from Chapter 7 onward rests on three ideas: an address, a variable that stores one, and memory that exists because the program asked for it while running. Book: Section 6.1.

Where We Left Off

Chapter 1 drew memory as numbered bytes; Chapters 2 and 3 searched and sorted arrays; Chapter 5 built stacks and queues on arrays and hit their weak spot: a **fixed size**.

We already wrote `&catalog[i]`, `const Book& b`, and passed arrays that the function could change. Today explains all of it.

Linked lists, trees and graphs keep their pieces **scattered** in memory and join them with **addresses**.

Running example: Pustaka Ceria's `Book` record (an id, a title, a writer and a year) with Laskar Pelangi, Filosofi Kopi, Bumi Manusia, Negeri 5 Menara and Cantik Itu Luka. Book: Section 6.1.

Memory as a Street of Numbered Houses

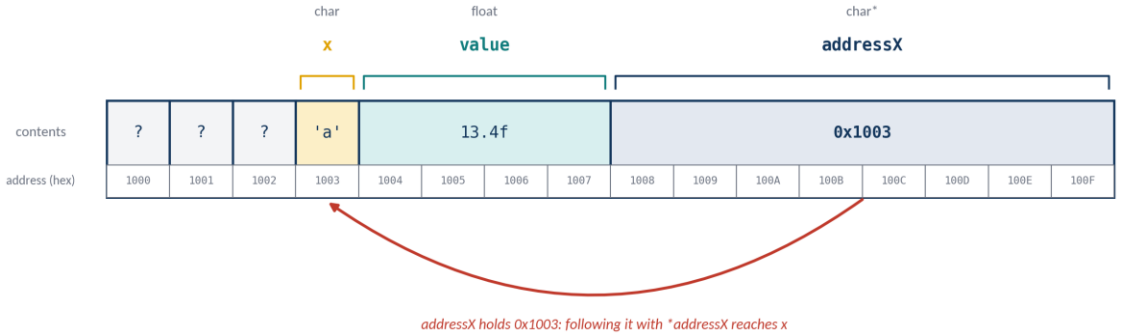


Figure 6.1 Sixteen bytes of memory. Each variable occupies consecutive bytes; the pointer `addressX` holds the address `0x1003`, so following it leads back to `x`.

Figure 6.1*: each byte has an address, written in hexadecimal;* `char x` *holds* `'a'` *at* `0x1003`*,* `float value` *fills* `0x1004`–`0x1007`*, and* `addressX` *holds the number* `0x1003`*: the address of* `x`*. Book: Section 6.2.1.*

What a Pointer Is

> **DEFINITION.** A *pointer* is a variable whose value is the address of a location in memory; it *points to* whatever lives there. A pointer that points to nothing holds *null*, written `nullptr`.

A friend's house number on a scrap of paper: the paper is not the house, but a way to reach it — and photocopying it builds no new house.

`&x` is the **address-of** operator: "the address of `x`".

`*p` in an expression **dereferences**: "the thing that lives at the address in `p`". Read on the right, write on the left.

`*&x` is `x`, and `&*p` is `p`: the two operators are inverses. Book: Sections 6.2.1–6.2.3.

A First Pointer

Listing 6.1 From `01_first_pointer.cpp`: a pointer to `float`, followed to read and to write.

```
    float value = 13.4f;
    float* address = &value;              // a pointer to float
    std::cout << "value = " << value << ", stored at " << address << '\n';
    std::cout << "*address = " << *address << '\n';

    *address = 27.5f;                     // write through the pointer
    std::cout << "after *address = 27.5f, value = " << value << '\n';
```

```
value = 13.4, stored at 0x7ffc2d719d94
*address = 13.4
after *address = 27.5f, value = 27.5
sizeof(char*) = 8, sizeof(float*) = 8, sizeof(double*) = 8
```

***Listing 6.1**: `*address = 27.5f` changes `value` itself — the pointer is a second route to the same `float`. Every pointer is 8 bytes on a 64-bit machine. Addresses change from run to run (address space layout randomization); the differences between them do not. Book: Section 6.2.2.*

Two Symbols, Three Meanings Each

Table 6.1 The three meanings of `*` and `&` in C++. Context always decides which one is meant.

Where it appears	* means	& means
In a declaration: `int* p;` / `int& r = a;`	`p` is a pointer to `int`	`r` is a reference to `int` (Section 6.9)
In an expression, before a pointer or variable: `*p` / `&a`	follow the pointer (dereference)	take the address of `a`
Between two operands: `a * b` / `a & b`	multiplication	bitwise AND

Table 6.2 An ordinary variable and a pointer side by side.

	Ordinary variable `int a;`	Pointer `int* p;`
Holds	a value of its type, such as 13	an address, such as `0x7ffc1645e8c4`, or `nullptr`
Typical operations	`+ - * /`, comparison	`&` to obtain an address, `*` to follow one, `+ -` to move within an array, comparison
Size	depends on the type (1, 4, 8, … bytes)	the same for every pointer type on a machine (8 bytes on 64-bit)
Lifetime of what it refers to	it *is* the object	may outlive, or be outlived by, the object it points to

***Table 6.1**: context decides. **Table 6.2**: the last row is the seed of every pointer bug — a pointer and the thing it points to have separate lives. C++ refuses to store a `long*` in an `int*`: the type decides how many bytes a dereference reads. Book: Sections 6.2.3 and 6.2.4.*

Copying an Address Versus Copying a Value

Listing 6.2 From `02_pointer_assign.cpp`: `*q = *p` copies a value between two boxes; `q = p` copies an address.

```
    // 3. Two pointers, two boxes: *q = *p copies a value between the boxes.
    int c = 3;
    int* q = &c;
    std::cout << "3. before *q = *p: *p = " << *p << ", *q = " << *q
              << ", p == q: " << (p == q) << '\n';
    *q = *p;
    std::cout << "   after  *q = *p: *p = " << *p << ", *q = " << *q
              << ", p == q: " << (p == q) << ", c = " << c << '\n';

    // 4. q = p: now q points at a as well, and c is no longer reachable via q.
    c = 3;
    q = p;
    std::cout << "4. after  q = p:   *p = " << *p << ", *q = " << *q
              << ", p == q: " << (p == q) << ", c = " << c << '\n';
```

```
3. before *q = *p: *p = 13, *q = 3, p == q: false
   after  *q = *p: *p = 13, *q = 13, p == q: false, c = 13
4. after  q = p:   *p = 13, *q = 13, p == q: true, c = 3
```

***Listing 6.2**: after `*q = *p`, `c` is 13 and the arrows have not moved; after `q = p`, both arrows reach `a`, and writing `*q = 99` changes `a`. Book: Section 6.3.1.*

Boxes and Arrows

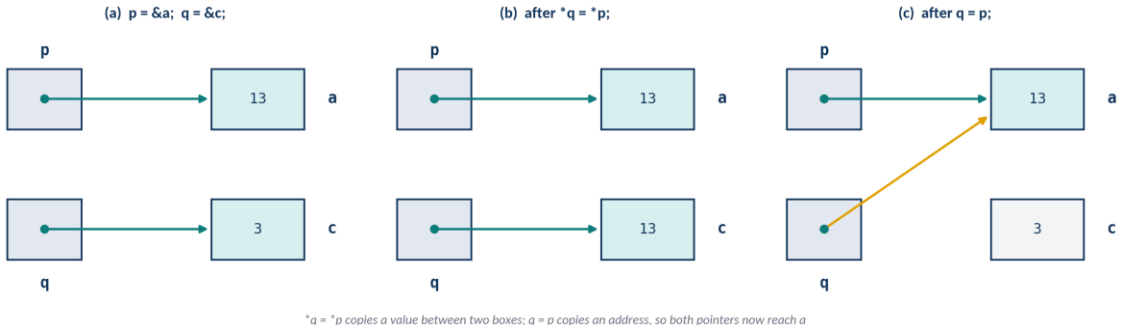


Figure 6.2 Two ways to assign with pointers. `*q = *p` moves a value between the boxes; `q = p` moves an arrow.

> **KEY INSIGHT.** Ask of every assignment with pointers: *is there a `*` on the left?* With a star, the object at the end of the arrow changes. Without, an arrow moves and no object changes.

***Figure 6.2**: draw every variable as a box, redraw only what a statement changes, and draw a new arrow when a pointer is copied. `p == q` asks "same object?", never "equal values?". Book: Sections 6.2.5, 6.3.1 and 6.3.2.*

Pointer Arithmetic

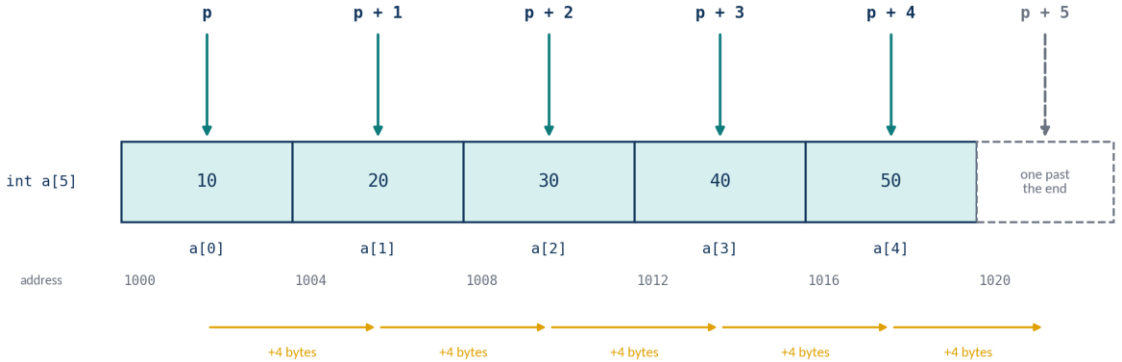


Figure 6.3 Pointer arithmetic on `int a[5]`. Each `+1` moves one element, four bytes; `p + 5` points one past the end, a position that may be computed and compared but never dereferenced.

$$\text{address}(p + i) = \text{address}(p) + i \cdot \text{sizeof}(T) \tag{6.1}$$

*Equation (6.1): adding 1 moves one element, not one byte — the address formula of Chapter 1 with a pointer as the base. `p + 5` is one past the end: it may be computed and compared, never dereferenced (**Figure 6.3**). Book: Section 6.3.3.*

Worked Example 1: Steps and Distances

$$q - p = \frac{\text{address}(q) - \text{address}(p)}{\text{sizeof}(T)} \tag{6.2}$$

```
char   (sizeof 1)  p+0: +0  p+1: +1  p+2: +2  p+3: +3
int    (sizeof 4)  p+0: +0  p+1: +4  p+2: +8  p+3: +12
double (sizeof 8)  p+0: +0  p+1: +8  p+2: +16  p+3: +24
Book   (sizeof 104)  p+0: +0  p+1: +104  p+2: +208  p+3: +312
*first = 10, *last = 50
*(first + 2) = 30, first[2] = 30
last - first = 4 elements
```

Equation (6.2): subtracting two pointers counts elements: `last - first` is 4, not 16 bytes. A `Book` step is 104 bytes: three 32-byte `std::string`s, a 4-byte `int`, padded to a multiple of 8 — and the compiler multiplies by `sizeof(Book)` so you never have to. Book: Section 6.3.3.

What Is Allowed, and What Is Not

Table 6.3 The operations C++ defines on a pointer `p` of type `T*`.

Operation	Example	Result	Condition
Take an address	`&a`	a `T*`	`a` is an object of type `T`
Dereference	`*p`, `p->m`	the object `p` points to	`p` points to a live object
Assign	`q = p`	`q` holds the same address	same pointer type
Compare for identity	`p == q`, `p != nullptr`	`bool`	always allowed
Order	`p < q`	`bool`	both into the same array
Move	`p + i`, `p - i`, `++p`, `--p`, `p += i`	a `T*`	result stays within the array or one past its end
Distance	`q - p`	`std::ptrdiff_t`, in elements	both into the same array
Subscript	`p[i]`	same as `*(p + i)`	as for move and dereference

UNDEFINED BEHAVIOR. No requirement at all on what happens: a crash, a plausible wrong answer, or perfect behavior until the next compiler upgrade. Violating any condition in the table is undefined behavior.

***Table 6.3**: adding two pointers, multiplying a pointer, or subtracting pointers into different arrays is rejected or undefined. Book: Section 6.3.4.*

An Array Name Becomes a Pointer

In almost every expression `a` decays to `&a[0]`; the exceptions are `sizeof(a)` and `&a`. The standard defines the brackets by Equation (6.3):

$$a[i] \equiv *(a + i) \tag{6.3}$$

Listing 6.3 From `04_array_walk.cpp`: the one-past-the-end loop, and `2[a]`.

```
    std::cout << "The value of a =";
    for (int* q = a; q != a + 5; ++q) {   // stop at one-past-the-end
        std::cout << ' ' << *q;
    }
    std::cout << "\n2[a] = " << 2[a] << " (the same as a[2] = " << a[2] << ")\n";
```

```
The value of a = 0 10 20 30 40
2[a] = 20 (the same as a[2] = 20)
```

Listing 6.3: $*(a+i) = *(i+a)$, so `i[a]` *is legal — seen once, it removes the magic from subscripts. The pair (first, one past the last) is a* ***half-open range*** $[first, last)$: `v.begin()` *and* `v.end()`. *Book: Sections 6.4.1 and 6.4.2.*

Strings Are Arrays of `char`

Listing 6.4 `05_danaya.cpp`: walking a C string forward and backward with a pointer; `overrun` repeats the classic mistake.

```
    char* p = s;                            // p = &s[0]
    std::cout << "Forward:  ";
    while (*p != '\0') {                    // stop at the terminator
        std::cout << *p;
        ++p;
    }
    std::cout << "\np now points at s[" << (p - s) << "], the terminator\n";

    std::cout << "Backward: ";
    while (p != s) {                        // walk back to the first char
        --p;
        std::cout << *p;
    }
```

```
sizeof(s) = 7, strlen(s) = 6
Forward:  Danaya
p now points at s[6], the terminator
Backward: ayanaD
```

***Listing 6.4**: `"Danaya"` is seven characters, six letters and `'\0'`. Walking again without resetting `p` reads past the array; built with `-fsanitize=address` and run as `./danaya overrun`, it reports a stack-buffer-overflow at line 32. Without the sanitizer it "works". Book: Section 6.4.3.*

Arrays as Function Parameters

An array passed to a function arrives as the address of its first element: the function can change the caller's array, and it **does not know the length**.

`int data[]` in a parameter list is rewritten to `int* data`; GCC warns if you ask for its `sizeof`.

```
Fill the data...
  in main: sizeof(data) = 40 (10 ints)
  inside fill: sizeof(data) = 8 (the size of a pointer)
sum of all ten = 55
sum of data[2..4] = 12
std::array sum = 26, std::vector sum = 10
```

Three ways to supply the length: a pointer and a count, a half-open range of two pointers (`sum(data + 2, data + 5)` is $3 + 4 + 5 = 12$), or a container passed by `const` reference. In a 2-D array `int g[][4]`, the row length must be in the type. Book: Sections 6.4.4 and 6.4.5.

The Arrow Operator, and a First Chain

> **DEFINITION.** For a pointer `p` to a structure, `p->m` means exactly `(*p).m`. (`*bp.title` would mean `*(bp.title)`: the dot binds tighter.)

Listing 6.5 `07_book_pointer.cpp`: the arrow operator on an array of `Book`, and a first chain of nodes.

```
    // Three nodes, each pointing at the next: a chain, not an array.
    Node third  {catalog[4], nullptr};
    Node second {catalog[0], &third};
    Node first  {catalog[2], &second};
    Node* head = &first;

    std::cout << "\nFollowing the chain from head:\n";
    for (Node* cur = head; cur != nullptr; cur = cur->next) {
        std::cout << "  " << cur->data.title << '\n';
    }
```

```
Following the chain from head:
  Bumi Manusia
  Laskar Pelangi
  Cantik Itu Luka
head->next->next->data.title = Cantik Itu Luka
```

***Listing 6.5**: three separate local variables, declared in reverse, form an ordered list — the order lives entirely in the `next` pointers. Start at the head, stop at null, advance by `next`. Book: Section 6.5.*

Nodes Linked by Pointers

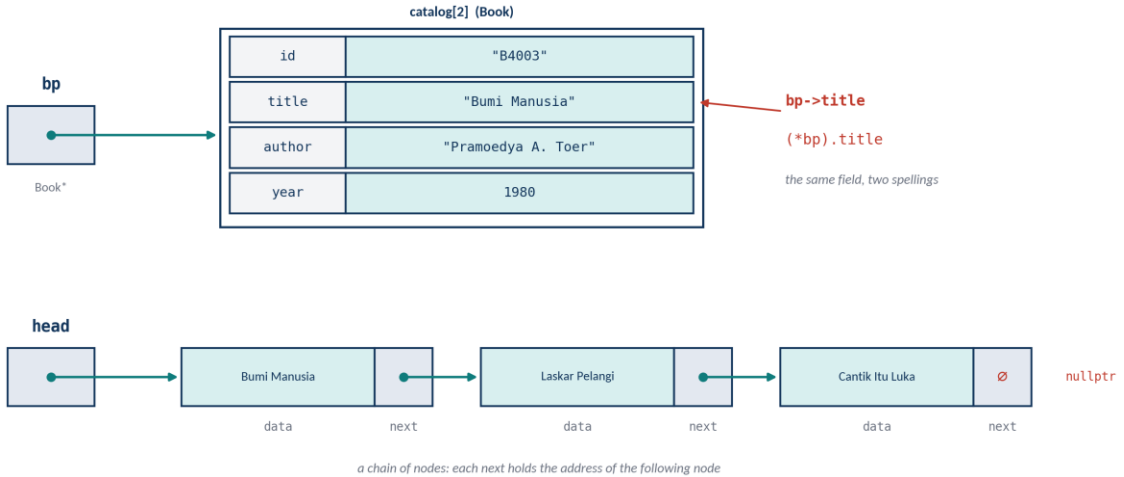


Figure 6.4 Top: a `Book*` and the two spellings of a field access. Bottom: three nodes linked by their `next` pointers, the shape of the linked lists of Chapter 7.

***Figure 6.4**: read `head->next->next->data.title` one arrow at a time: Bumi Manusia, Laskar Pelangi, then Cantik Itu Luka. Change one pointer and the order changes without moving a single `Book`: the whole idea of Chapter 7. Book: Section 6.5.*

Two Places to Live: the Stack and the Heap

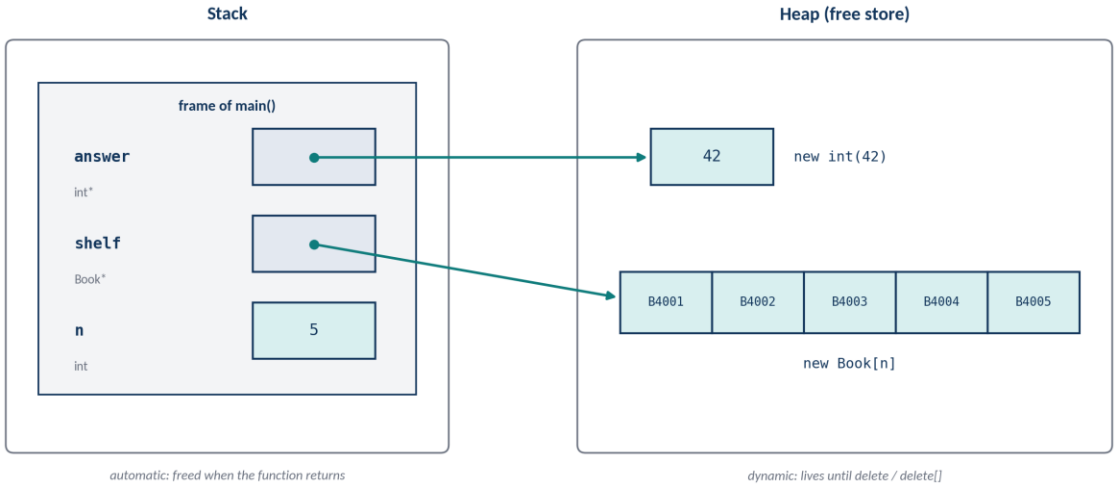


Figure 6.5 Automatic variables live in the stack frame of their function; objects created with `new` live on the heap and are reached only through pointers.

> **DEFINITION.** A *dynamically allocated* object is created by `new T` (or `new T[n]`) while the program runs and lives until `delete` (or `delete[]`), whichever function created it. The pointer returned by `new` is the only way to reach it.

***Figure 6.5**: automatic variables live in their function's frame; heap objects outlive it. This "heap" has nothing to do with the heap of Chapter 14. Book: Section 6.6.1.*

`new`, `delete`, and a Doubling Array

Listing 6.6 From `08_dynamic.cpp`: when the heap block is full, allocate twice the capacity, copy, and give the old block back.

```
    void push(int value) {
        if (size == capacity) {
            int newCapacity = (capacity == 0) ? 1 : 2 * capacity;
            int* bigger = new int[newCapacity];
            for (int i = 0; i < size; ++i) {
                bigger[i] = data[i];      // copy the old elements across
            }
            copies += size;
            delete[] data;                // give the old block back
            data = bigger;
            capacity = newCapacity;
            std::cout << "  grow: capacity " << capacity << ", copied " << size << '\n';
        }
        data[size++] = value;
    }
```

```
  grow: capacity 32, copied 16
size 17, capacity 32, last 289, total copies 31 (< 2 * 17 = 34)
```

***Listing 6.6**: every `new` pairs with one `delete`, every `new[]` with one `delete[]`; `~IntArray()` frees the last block — the seed of smart pointers. After `delete answer;` the program sets `answer = nullptr;`. If `new` fails it throws `std::bad_alloc`. Book: Sections 6.6.2 and 6.6.4.*

The Doubling Rule

Pseudocode 6.1 `DYNAMIC-ARRAY-PUSH`: append `x`, first doubling the capacity when the array is full.

```
DYNAMIC-ARRAY-PUSH(A, x)
1  if A.size == A.capacity
2      if A.capacity == 0
3          newCap = 1
4      else newCap = 2 · A.capacity
5      B = allocate an array of newCap elements
6      for i = 0 to A.size − 1
7          B[i] = A.data[i]
8      free A.data
9      A.data = B
10     A.capacity = newCap
11 A.data[A.size] = x
12 A.size = A.size + 1
```

$$C(n) = 1 + 2 + 4 + \cdots + 2^k = 2^{k+1} - 1 < 2n \tag{6.4}$$

Pseudocode 6.1 *and Equation (6.4): 17 pushes cost 31 copies, below $2 \cdot 17 = 34$: fewer than two per push, amortized $O(1)$. Growing by a fixed $+10$ would cost $\Theta(n^2)$. Book: Section 6.6.3.*

Growth Moves the Elements

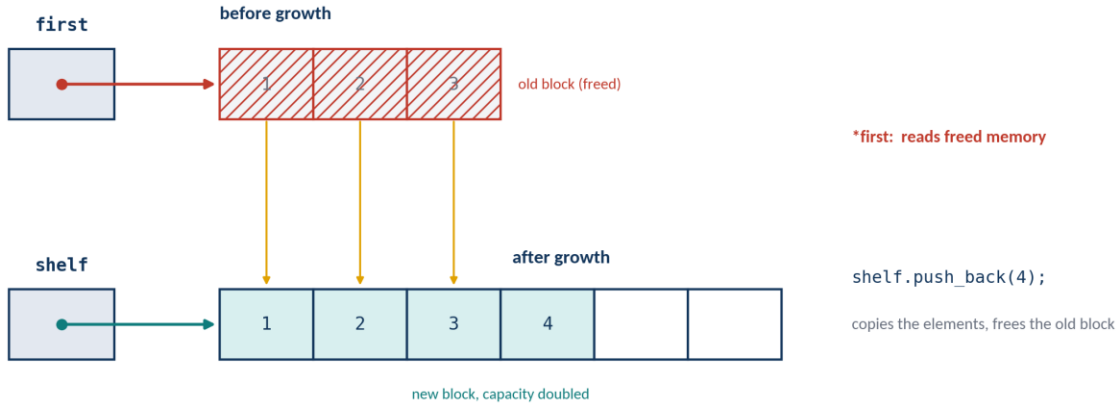


Figure 6.6 One growth step of a doubling array. The elements are copied into a new block and the old block is freed, so a pointer taken before the growth now points into freed memory.

***Figure 6.6**: after one growth step the old block is freed, so a pointer taken before the growth now points into freed memory — the `stale` bug we are about to catch. Book: Section 6.6.3.*

Five Bugs, One Root Cause

Dangling pointer: the object is gone, the address remains — *use after free*.

Memory leak: the object remains, no pointer does. **Double free:** the same block deleted twice.

Mismatched delete: `new[]` freed with `delete`. **Null dereference:** a “not found” result trusted blindly.

Plus the **out-of-bounds access** of `"Danaya"`.

Every one is a disagreement between a pointer’s life and its object’s life. Book: Section 6.7.1.

Three Lifetime Bugs, Drawn

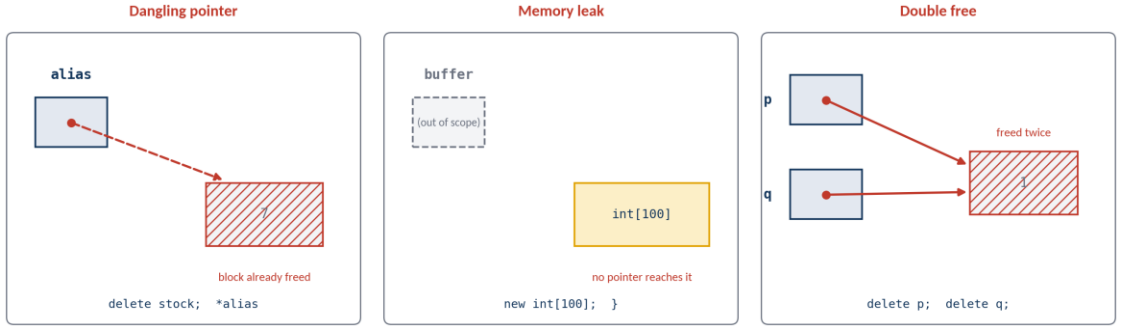


Figure 6.7 Three lifetime bugs drawn as boxes and arrows. Hatched blocks have already been freed.

***Figure 6.7**: an arrow into a hatched (freed) block, a block with no arrow into it, and two arrows into a block freed twice. The drawing makes each bug visible at once. Book: Section 6.7.1.*

Not Academic: the 2025 CWE Top 25

Table 6.4 Memory-safety weaknesses in the 2025 CWE Top 25 Most Dangerous Software Weaknesses, published by MITRE in December 2025.

2025 rank	Weakness (CWE identifier)	In this chapter
5	Out-of-bounds Write (CWE-787)	`overflow` in **Listing 6.7**
7	Use After Free (CWE-416)	`dangling`, `stale` in **Listing 6.7**
8	Out-of-bounds Read (CWE-125)	`overrun` in **Listing 6.4**
11	Buffer Copy without Checking Size of Input (CWE-120)	the reason for `strlen`-style length checks
13	NULL Pointer Dereference (CWE-476)	`null` in **Listing 6.7**
16	Heap-based Buffer Overflow (CWE-122)	`overflow` in **Listing 6.7**

***Table 6.4**: six of the 25 most dangerous weaknesses of 2025 are exactly the bugs of this chapter. Book: Section 6.7.1.*

The Bugs, on Purpose

Listing 6.7 From `09_mem_bugs.cpp`: a use after delete, and a pointer that goes stale when a vector grows.

```
void useAfterDelete() {
    std::vector<int*> owners;           // two records of the same block
    owners.push_back(new int(7));
    owners.push_back(owners[0]);        // a second pointer, same address
    delete owners[0];                   // the block is returned ...
    std::cout << "stock is " << *owners[1] << '\n';  // ... but is still used
}

void staleAfterGrowth() {
    std::vector<int> shelf = {1, 2, 3};
    int* first = &shelf[0];             // points into the vector's block
    shelf.push_back(4);                 // may move every element elsewhere
    std::cout << "first is " << *first << '\n';
}
```

```
usage: mem_bugs dangling|stale|leak|double|mismatch|overflow|null
no bug chosen; the correct versions:
stock is 7
```

***Listing 6.7**: each bug is chosen by a word on the command line. Compiled normally, most of them appear to work: `./mem_bugs dangling` printed `stock is 7`. The program is wrong, the output is right. Book: Section 6.7.2.*

AddressSanitizer Reads Like a Detective

```
==23665==ERROR: AddressSanitizer: heap-use-after-free on address 0x502000000010 ...
READ of size 4 at 0x502000000010 thread T0
    #0 0x55bc1c95b8eb in useAfterDelete() 09_mem_bugs.cpp:14
    #1 0x55bc1c95cb8a in main 09_mem_bugs.cpp:84
0x502000000010 is located 0 bytes inside of 4-byte region [0x502000000010,0x502000000014)
freed by thread T0 here:
    #0 0x7f30524ff5e8 in operator delete(void*, unsigned long)
    #1 0x55bc1c95b864 in useAfterDelete() 09_mem_bugs.cpp:13
```

What: heap use after free. **Which access:** a 4-byte read. **Where:** line 14. **Who freed it:** the `delete` on line 13.

The `stale` report: nobody called `delete` — the block was freed *inside* `push_back`. Pointers, references and iterators into a `std::vector` die when it grows.

Build with `g++ -std=c++17 -g -fsanitize=address,undefined`: about twice as slow, and every bug is reported at its exact line. Valgrind runs the unchanged program 20–50 times slower and gives the best leak reports ("definitely lost"). Book: Sections 6.7.3 and 6.7.4.

Which Tool, on Which Computer

Table 6.5 AddressSanitizer and Valgrind compared, as of September 2026.

	AddressSanitizer	Valgrind (Memcheck)
How it works	compiler inserts checks; rebuild with `-fsanitize=address`	runs the unchanged binary on a simulated CPU
Slowdown	about 2×	about 20–50×
Linux	GCC and Clang	yes
macOS	Apple Clang (`clang++ -fsanitize=address`)	not on Apple silicon
Windows	Microsoft C++ (`cl /fsanitize=address`) or Clang; not in the MinGW g++ of MSYS2	no (use WSL, a Linux environment inside Windows)
Best at	out-of-bounds and use-after-free, with exact lines	leak reports; checking programs you cannot rebuild

KEY INSIGHT. Tools find bugs that have *happened* in a run; they cannot prove a program correct. Give every heap object one clear owner, prefer containers and smart pointers, and let a destructor do the freeing.

***Table 6.5**: C++26, completed in March 2026, makes reading an uninitialized local erroneous rather than undefined and adds a hardened library; GCC 16.1 implements much of it. None of that checks raw pointers. Book: Section 6.7.5.*

`nullptr`: a Pointer to Nothing

Listing 6.8 From `10_nullptr.cpp`: a search that returns the address of what it found, or `nullptr`.

```
const std::string* findTitle(const std::string* titles, int n, char firstLetter) {
    for (int i = 0; i < n; ++i) {
        if (!titles[i].empty() && titles[i][0] == firstLetter) {
            return &titles[i];            // found: the address of the element
        }
    }
    return nullptr;                       // not found: the null pointer
}
```

```
Calling describe with 0 and with nullptr:
  describe(int 0)
  describe(int* null)
p == nullptr: true, p == q: true, !p: true
B: found "Bumi Manusia" at index 2
Z: no title starts with this letter
```

Listing 6.8: *`0` is an `int`, so `describe(0)` picks the `int` overload; `nullptr` converts only to pointers.* ***Check before you dereference***: *`if (hit != nullptr)`. A pointer parameter says "may be null"; a reference says "never null". Book: Section 6.8.*

References: a Second Name

DEFINITION. `T& r = x;` makes `r` an alias of `x`: every use of `r` is a use of `x`. A reference must be initialized, can never be re-seated, and can never be null.

Listing 6.9 From `11_references.cpp`: assigning to a reference assigns to the object; `r = b` does not re-seat it.

```
    int a = 13;
    int& r = a;                          // r is another name for a
    r = 20;
    std::cout << std::boolalpha;
    std::cout << "after r = 20: a = " << a << ", &r == &a: " << (&r == &a) << '\n';

    int b = 99;
    r = b;                               // assigns 99 to a; r is NOT re-seated
    b = 7;
    std::cout << "after r = b; b = 7: a = " << a << ", b = " << b
              << ", r = " << r << '\n';
```

```
after r = 20: a = 20, &r == &a: true
after r = b; b = 7: a = 99, b = 7, r = 99
```

***Listing 6.9**: after `r = b; b = 7;`, `a` and `r` stay 99 — there is no arrow to move, only a second label glued to the same box. `for (int& x : stock)` changes the container in place. Book: Section 6.9.1.*

References Versus Pointers

Table 6.6 Pointers and references compared.

Property	Pointer `T* p`	Reference `T& r`
Must be initialized	no (but should be)	yes
Can be null	yes, `nullptr`	no
Can be re-seated to another object	yes, `p = &other;`	no; `r = other;` assigns a value
Syntax to reach the object	`*p`, `p->m`	`r`, `r.m`
Arithmetic	yes, within an array	no
Has its own address	yes, `&p` is a `T**`	`&r` is the object's address
Typical use	optional objects, re-pointable links (list nodes), arrays, ownership	parameters, loop variables, aliases

***Table 6.6**: use a reference when there must always be an object and it will not change; use a pointer when "no object" is possible or the pointer must later point elsewhere. Linked structures need pointers for both reasons. Book: Section 6.9.2.*

Three Ways to Pass an Argument

By value `T x`: the parameter is initialized with a **copy** of the argument.

By pointer `T* x`: the parameter holds a copy of an **address**, supplied with `&`.

By reference `T& x`: no new object — the parameter is **another name for the argument**.

```
a in passByValue() is 26
a in main() after passByValue() is 13
*a in passByPointer() is 26
a in main() after passByPointer() is 26
```

A declaration (prototype) promises a function; formal parameters are initialized from actual ones; locals live only while their block runs, while `static` locals keep their values between calls. Book: Sections 6.10.1 and 6.10.2.

Worked Example 2: the Swap Story

Listing 6.10 `13_swap.cpp`: swapping by value (fails), by pointer and by reference (both work), and `std::swap`.

```
void swapByValue(int x, int y) {          // x, y are copies
    int t = x;
    x = y;
    y = t;
    std::cout << "  inside swapByValue:     x = " << x << ", y = " << y << '\n';
}

void swapByPointer(int* x, int* y) {      // x, y hold the caller's addresses
    if (x == nullptr || y == nullptr) return;
    int t = *x;
    *x = *y;
    *y = t;
}

void swapByReference(int& x, int& y) {    // x, y ARE the caller's variables
    int t = x;
    x = y;
    y = t;
}
```

***Listing 6.10**: the best experiment in the whole topic. The pointer version checks for `nullptr`; the reference version looks at the call site exactly like the by-value one. Book: Section 6.10.3.*

The Swap Story, Traced

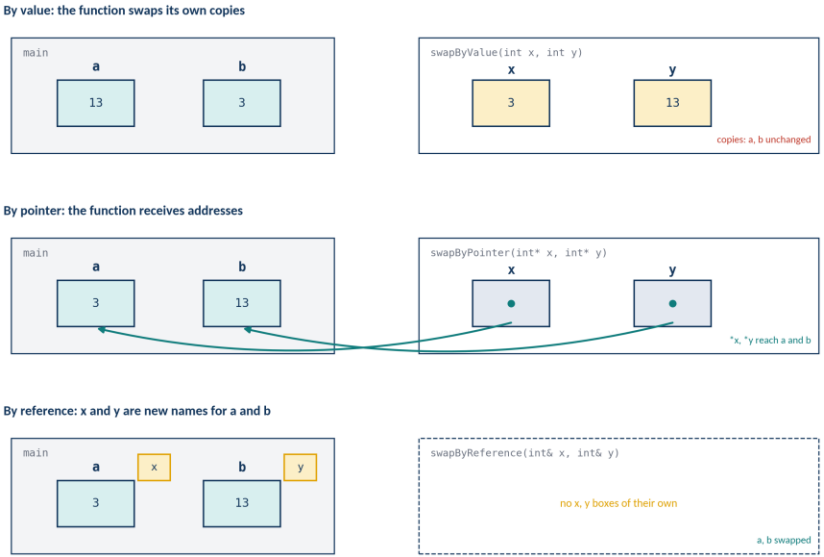


Figure 6.8 The same swap written three ways. By value the function exchanges its own copies; by pointer it exchanges the caller's variables through their addresses; by reference its parameters are the caller's variables under new names.

Figure 6.8*: by value, the copies are swapped and vanish; by pointer,* $a = 3, b = 13$*; by reference they are exchanged again,* 13 *and* 3*;* `std::swap` *makes them* 3 *and* 13*. A function called by value cannot damage its caller's variables. Book: Section 6.10.3.*

The Cost of Passing by Value, and a Choice

$$\text{bytes copied} = k \cdot s \qquad \text{by value,} \qquad 0 \qquad \text{by reference or pointer} \tag{6.5}$$

Table 6.7 Choosing how a C++ function receives its argument.

The function needs to…	Write the parameter as	The call looks like	Example
read a small value (`int`, `double`, `char`, a pointer)	`T x`	`f(a)`	`add(int x, int y)`
read a large object	`const T& x`	`f(book)`	`print(const Book& b)`
change the caller's object, which always exists	`T& x`	`f(book)`	`swapByReference(int& x, int& y)`
change an object that may be absent, or accept an array	`T* x` (check for `nullptr`)	`f(&book)`, `f(array)`	`swapByPointer`, `fill(int* data, int n)`
keep its own copy anyway	`T x`	`f(book)`	`makeReprint` (Section 6.12)
take over a heap object	`std::unique_ptr<T> x`	`f(std::move(p))`	`withdraw` (Section 6.13)

*Equation (6.5): k calls copy $k \cdot s$ bytes by value, plus any heap memory the object owns. **Table 6.7**: small built-ins by value, large objects by `const T&`, changes by `T&`, optional objects by `T*`. Book: Sections 6.10.4 and 6.10.5.*

`const` Correctness

Table 6.8 The four combinations of `const` with a pointer.

Declaration	Read right to left	Change *p?	Change p?
`int* p`	`p` is a pointer to int	yes	yes
`const int* p`	`p` is a pointer to int that is const	no	yes
`int* const p`	`p` is a const pointer to int	yes	no
`const int* const p`	`p` is a const pointer to const int	no	no

```
1000 calls by value:     1000 copies
1000 calls by const ref: 0 copies
```

***Table 6.8**: read right to left from the name. A const member function, `int age(int now) const`, may be called on a `const Book`. The copy counter confirms Equation (6.5): 1,000 copies by value, none by `const&`. Break a promise and the compiler stops you. Book: Section 6.11.*

Returning Pointers and References Safely

Listing 6.11 From `15_find_book.cpp`: returning a pointer into the caller's array, or `nullptr`.

```
// Returns the address of the matching element, or nullptr. The array
// belongs to the caller, so the address stays valid after we return.
const Book* findById(const Book* first, const Book* last, const std::string& id) {
    for (const Book* p = first; p != last; ++p) {
        if (p->id == id) return p;
    }
    return nullptr;
}
```

Safe: a pointer or reference into storage the caller owns; a new value (return value optimization); a `std::unique_ptr` for a new heap object.

Never: the address of, or a reference to, a local variable — GCC 13 warns and even returns null.

Listing 6.11 *is the pattern of every search in a linked structure: the result is a pointer to a node, or null. Book: Section 6.12.*

Smart Pointers: Ownership in the Type

RAII: tie every resource to an object whose destructor releases it, on normal exit, early `return` or exception alike.

`std::unique_ptr<T>`: exactly one owner; cannot be copied, only **moved** with `std::move`; costs nothing over a raw pointer.

```
1. make_unique
  [create B4001]
  a->title = Laskar Pelangi
2. move ownership from a to b
  a is empty: true, b->id = B4001
3. hand b to a function that keeps it
  withdrawing Laskar Pelangi
  [destroy B4001]
  back in main, b is empty: true
```

`withdraw(std::move(b))` takes ownership, so `[destroy B4001]` prints inside step 3. A vector of `unique_ptr` owns a shelf; `get()` gives a non-owning observer; `make_unique<int[]>` uses `delete[]`. No `delete` anywhere. Book: Sections 6.13.1 and 6.13.2.

Shared Owners, and a Cycle

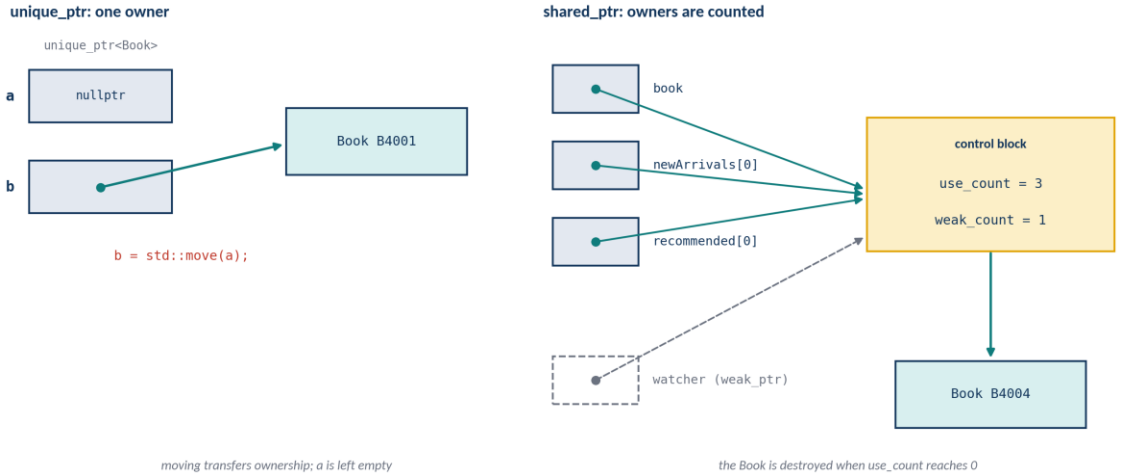


Figure 6.9 Left: a `unique_ptr` has one owner, and moving transfers it. Right: three `shared_ptr`s share one `Book` through a control block that counts them; a `weak_ptr` observes without being counted.

***Figure 6.9**: a `shared_ptr` counts its owners in a control block — 1, 3, 2, 1, then 0 and `[destroy B4004]`; a `weak_ptr` observes without counting. Two members holding each other by `shared_ptr` leak; linking them by `weak_ptr` breaks the cycle. Owners point down, back-links point up. Book: Section 6.13.3.*

Which Pointer When

Table 6.9 Raw pointers, references and smart pointers, and when to use each.

Tool	Owns?	Copyable?	Cost	Use it for
`T*`	no (by convention)	yes	none	observing, optional arguments, links to objects owned elsewhere
`T&`	no	binds once	none	always-present arguments and aliases
`std::unique_ptr<T>`	yes, alone	move only	none	the default owner of a heap object
`std::shared_ptr<T>`	yes, shared	yes (counted)	control block, atomic count updates	objects with several owners of unknown lifetime
`std::weak_ptr<T>`	no	yes	uses the control block	back-links, caches, breaking cycles

KEY INSIGHT. Chapters 7 to 13 still write raw `new` and `delete`, because a first study of data structures is where you learn what smart pointers do for you. In production, a list node would hold its successor in a `unique_ptr`.

***Table 6.9**: a raw pointer remains fine as a non-owning observer; what modern code avoids is a raw pointer that owns. Book: Section 6.13.3.*

Java References and Python Names

Java: primitives hold values; class and array variables hold **references**; every argument is **passed by value** — for an object, the value is the reference.

Python: every variable is a **name bound to an object**; a call binds the parameter to the caller's object — *call by sharing*.

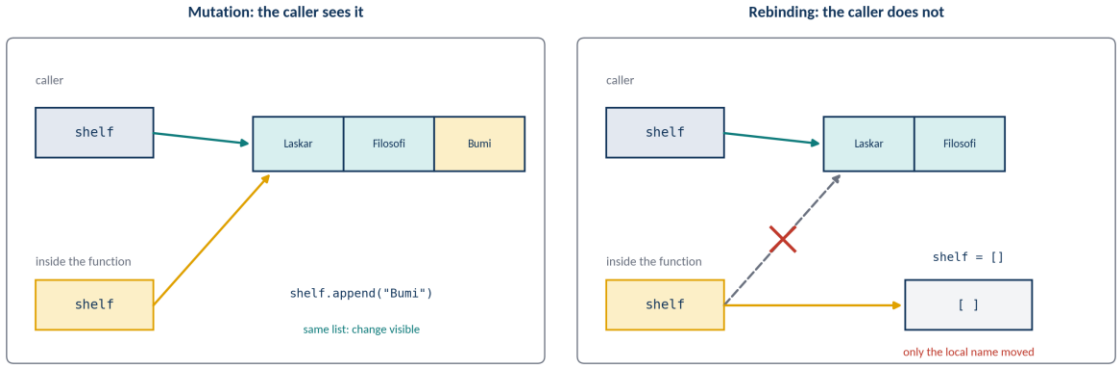


Figure 6.10 Left: the function mutates the list its parameter refers to, and the caller sees the change. Right: the function rebinds its parameter to a new list, and the caller's name still refers to the old one.

***Figure 6.10**: the callee can **mutate** the shared object, and the caller sees it; it can **rebind** its own name, and the caller does not. Book: Sections 6.14.1 and 6.14.2.*

Mutation Versus Rebinding in Python

Listing 6.12 From `refs_demo.py`: mutation, rebinding, an immutable `int`, and the mutable default argument.

```
def add_book(shelf):
    shelf.append("Bumi Manusia")       # mutates the caller's list

def replace_shelf(shelf):
    shelf = []                         # rebinds the local name only
    shelf.append("Negeri 5 Menara")

def bump_year(year):
    year += 1                          # ints are immutable: a new int, local name

def add_tag(tag, tags=[]):             # the default list is created ONCE
    tags.append(tag)
    return tags
```

```
shelf after add_book:      ['Laskar Pelangi', 'Filosofi Kopi', 'Bumi Manusia']
shelf after replace_shelf: ['Laskar Pelangi', 'Filosofi Kopi', 'Bumi Manusia']
a is b: False, a == b: True
add_tag('novel')  -> ['novel']
```

***Listing 6.12**: `year += 1` makes a new `int` and rebinds a local name. The default `tags=[]` is created once, so the second call returns `['novel', 'sejarah']`; use `tags=None`. Java behaves the same: `==` on objects compares references, `equals` compares contents. Book: Sections 6.14.2 and 6.14.3.*

The Swap Story in Python

Listing 6.13 From `swap_demo.py`: rebinding local names swaps nothing; mutating two shared lists does.

```python
def swap_ints(x, y):
    x, y = y, x                         # rebinds the local names only

def swap_boxes(x, y):
    x, y = y, x                         # still only local names

def swap_contents(x, y):
    x[0], y[0] = y[0], x[0]             # mutates the two shared lists
```

```
after swap_ints:     a = 13, b = 3
after swap_boxes:    box_a[0] = 13, box_b[0] = 3
after swap_contents: box_a[0] = 3, box_b[0] = 13
after a, b = b, a:   a = 3, b = 13
```

Listing 6.13*: neither Java nor Python can swap two of the caller's variables. The idiom is* `a, b = b, a` *in the caller's own scope; Java sorts with* `swapInArray(a, i, j)`*. CPython frees by reference counting plus a cycle collector; Java by a tracing garbage collector. Book: Sections 6.14.4 and 6.14.5.*

Three Languages Side by Side

Table 6.10 Pointers, references and parameter passing in C++, Java and Python.

Question	C++	Java	Python
What does a variable hold?	the object itself, or a pointer, or is a reference	a primitive value, or a reference to an object	a binding of a name to an object
Can you see or compute addresses?	yes: `&`, `*`, pointer arithmetic	no	no (`id` gives an identity number)
How are arguments passed?	by value, by pointer or by reference: your choice per parameter	always by value; for objects the value is a reference	the parameter is bound to the argument's object ("call by sharing")
Can a function swap two caller variables?	yes, by pointer or reference	no	no (use `a, b = b, a`)
Can a function change a caller's object?	yes, via pointer or non-const reference	yes, if the object is mutable	yes, if the object is mutable
Same object? / Same value?	`p == q` / `*p == *q`	`a == b` / `a.equals(b)`	`a is b` / `a == b`
"No object"	`nullptr`	`null`	`None`
Using "no object"	undefined behavior (usually a crash)	`NullPointerException`	`AttributeError` or `TypeError`

Question	C++	Java	Python
Freeing heap objects	`delete`, or a smart pointer's destructor	tracing garbage collector	reference counting plus cycle collector

Table 6.10 (Continued) *Pointers, references and parameter passing in C++, Java and Python.*

Question	C++	Java	Python
Dangling pointer or double free possible?	yes	no	no
Leaks possible?	yes	yes, by keeping unneeded references	yes, by keeping unneeded references

Table 6.10*: garbage collection removes dangling pointers and double frees, which is why Java and Python are memory-safe; it does not remove leaks through references kept by mistake. Book: Section 6.14.6.*

Profitina in Practice

A scan is a pipeline of stages that all receive *the same* result object: sharing is free, copying a page's text at every stage would cost Equation (6.5).

Sharing brings an obligation: a read-only stage must not mutate; the release gate of well over a thousand tests checks exactly that.

One expensive object, many users: the local language model is loaded once and shared by reference.

"No model" is handled explicitly, never dereferenced blindly: the `nullptr` discipline at the scale of a service.

The mutable default argument is a classic source of such bugs in Python web services: one list shared by every request. Book: Section 6.16.

Profitina Laundry: Moving Ownership

Listing 6.14 From `18_laundry.cpp`: promotion moves the owning pointer between queues; the order itself never moves.

```
bool promote(Queue& from, Queue& to, int number) {
    for (auto it = from.begin(); it != from.end(); ++it) {
        if ((*it)->number == number) {
            to.push_back(std::move(*it));  // ownership changes hands
            from.erase(it);                // erase the now-empty slot
            return true;
        }
    }
    return false;
}
```

```
regular:  #101 Sari Rp21000  #102 Joko Rp38500  #103 Rina Rp14000
regular:  #101 Sari Rp21000  #103 Rina Rp14000
express:  #102 Joko Rp38500
watched still sees #102 (same object, new owner)
quote with 10% off: Rp34650, stored price still Rp38500
  [order 102 released]
```

***Listing 6.14**: the raw observer `watched` still sees #102 after the move. `quoteWithDiscount` works on a copy (Rp34650, stored price still Rp38500); `applyDiscount` takes a reference and changes the real price. No `delete`, and Valgrind reports all blocks freed. Book: Section 6.16.*

Common Mistakes

`q = p` for `*q = *p`; `int* p, q;` declares one pointer; an uninitialized pointer; no null check.

Walking past the end; `sizeof` of an array parameter; returning the address of a local.

Forgetting `delete`, deleting twice, `delete` for `new[]`; a pointer into a vector after it grows.

Expecting `r = b` to re-seat a reference; passing large objects by value; `shared_ptr` cycles.

Java `==` on strings; expecting a method to rebind the caller's variable; Python mutable defaults.

Fifteen mistakes, each with its fix, are listed in the book. Book: Section 6.17.

Summary: Five Things to Remember

1. A pointer holds an address: `&` takes one, `*` follows one; `p + i` moves i elements (Equation (6.1)), and `a[i]` is `*(a + i)` (Equation (6.3)).
2. `new` and `delete` pair exactly; a doubling array grows in amortized $O(1)$ (Equation (6.4)); ASan and Valgrind catch the five lifetime bugs.
3. `nullptr` means "no object"; a reference is an alias that is never null and never re-seated.
4. By value copies, by pointer copies an address, by reference aliases; large read-only arguments are `const T&` (Equation (6.5)).
5. `unique_ptr` owns alone, `shared_ptr` counts, `weak_ptr` observes; Java and Python can mutate a shared object but never rebind the caller's variable.

Reaching element i of an array is $\Theta(1)$; reaching the i-th node of a chain is $\Theta(i)$ — and inserting into a chain is two pointer changes. Book: Sections 6.15 and 6.18.

Next Time

Chapter 7 — Singly Linked Lists. Nodes created while the program runs, joined by `next` pointers: insertion without shifting.

Insert and delete at the front, at the back and in the middle; traverse, search, reverse.

Before then: work Exercises 6.3, 6.6, 6.10 and 6.12, and run one bug of `09_mem_bugs.cpp` under AddressSanitizer.

The loop `for (Node* cur = head; cur != nullptr; cur = cur->next)` of today is the loop you will write most often next week. Book: Sections 6.19 and 6.20.

Lecture 7 — Singly Linked Lists

Course: Data Structure (EF234201)

Book: Irfan Subakti, *Data Structure with C++, Java and Python*, Chapter 7

Lecturer: Irfan Subakti — yifana@gmail.com

Today we build the first data structure made entirely of pointers: nodes created while the program runs, joined by `next` fields, reached from one pointer called `head`. Book: Chapter 7, Sections 7.1–7.16.

What This Meeting Covers

1. **The idea** — a treasure hunt, its history (IPL, COMIT, LISP), and the vocabulary: node, `next`, `head`, NULL.
2. **Array versus list** — random against sequential access, the cost of shifting, the memory per element.
3. **The head-pointer list** — insert and delete at both ends, traverse, clear; search, insert in the middle, delete by key, reverse.
4. **Variants** — a dummy head node, a pointer to the link, head and tail, the circular list, linked stacks and queues.
5. **Proving it safe and choosing well** — Valgrind, AddressSanitizer, the standard libraries, a benchmark, Profitina.

Every operation today is drawn before and after, then coded; the order of two pointer assignments will matter again and again. Book: Section 7.1.

Where We Left Off

Chapter 5 built a stack and a queue on an **array**: fast, but with a maximum size guessed before the first element arrived.

Chapter 6 gave us the tool that removes the guess: `new` hands back the address of fresh memory, `delete` returns it, `->` reaches a field, and a reference parameter lets a function change the caller's pointer.

Today these tools come together: each element lives in its own **node**, and every node remembers where the next one is.

The price: a list cannot jump to its fifth element, each node pays for a pointer, and one misplaced line can lose a whole chain. We meet each problem on purpose, draw it, measure it and fix it. Book: Section 7.1.

A Treasure Hunt

The first clue at the gate: “Look under the big mango tree.” There lie a prize and the next clue, and so on until “The hunt is over.”

Each stop holds two things: a prize (the data) and directions to the next stop (the pointer).

The stops can be anywhere — not side by side, not in order.

You can only follow the chain from the start: the fifth prize needs the first four clues.

Adding or removing a stop changes one clue; nobody moves a prize.

The card handed out at the gate is a pointer variable, `head`. The idea dates from 1955–1956: Newell, Shaw and Simon’s IPL at RAND, then Yngve’s COMIT and McCarthy’s LISP. Book: Sections 7.2.1 and 7.2.2.

The Vocabulary

DEFINITION. A *linked list* is a sequence of *nodes*; every node stores one element (its *data*) and a pointer, its *link* or `next` field, to the following node. The list is reached through an external pointer to its first node, `head`. The last node's `next` is null (`nullptr`, `null`, `None`); an empty list has a null `head`.

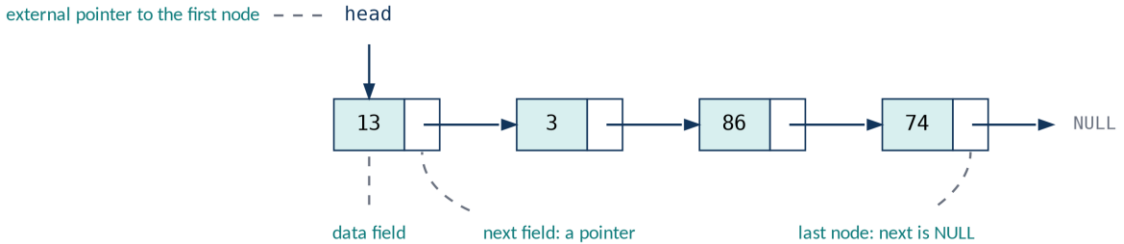


Figure 7.1 A singly linked list holding 13, 3, 86 and 74. Every node has a data field and a `next` field; `head` points at the first node, and the last node's `next` is NULL.

***Figure 7.1**: data on the left, `next` on the right. Singly linked: one forward link, no way back (Chapter 9 adds one). Non-circular: the chain ends at NULL, the stop signal of every loop today. Book: Section 7.2.3.*

Two Ways to Keep Five Numbers

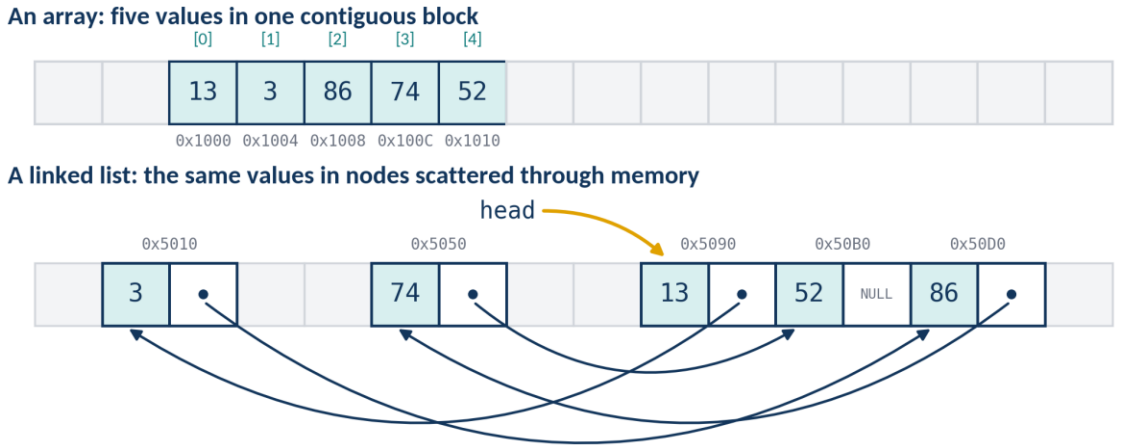


Figure 7.2 The same five values in an array (one contiguous block, consecutive addresses) and in a linked list (nodes wherever the allocator put them, joined by `next` pointers). The addresses are illustrative.

***Figure 7.2**: the array is one block with consecutive addresses; the list's nodes sit wherever `new` placed them — 13 at `0x5090`, 3 at `0x5010`, 86 at `0x50D0` — joined only by `next`. Book: Section 7.3.1.*

Random Access Against Sequential Access

$$\text{address}(i) = b + i \cdot s \tag{7.1}$$

$$T_{\text{reach}}(k) = k \text{ pointer steps}, \qquad 0 \le k \le n-1 \tag{7.2}$$

$$\frac{1}{n}\sum_{k=0}^{n-1} k = \frac{1}{n} \cdot \frac{(n-1)n}{2} = \frac{n-1}{2} \tag{7.3}$$

$$0 + 1 + 2 + \cdots + (n-1) = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} \tag{7.4}$$

Equation (7.1): `a[3]` in $O(1)$. Equation (7.2), Equation (7.3): node k needs k steps, $(n-1)/2$ on average: $\Theta(n)$. Equation (7.4): building an array by front insertion moves $\Theta(n^2)$ elements; a list needs two pointer changes per insertion. Book: Section 7.3.1.

The Comparison, Feature by Feature

Table 7.1 Arrays and singly linked lists compared.

Feature	Array	Linked list
Memory layout	One contiguous block	Separate nodes anywhere, joined by pointers
Size	Fixed when created (a `std::vector` regrows by copying)	Grows and shrinks one node at a time
Access to element k	Random access, $O(1)$, by Equation (7.1)	Sequential access, $\Theta(k)$ steps, by Equation (7.2)
Insert or delete at the front	$\Theta(n)$: every element shifts	$O(1)$: two pointer changes
Insert or delete after a known position	$\Theta(n)$ shifts in the worst case	$O(1)$
Extra memory per element	None	One pointer (8 bytes on a 64-bit machine) plus allocator bookkeeping
Releasing the structure	One `delete[]` for the whole block	Every node must be released, one by one
Cache friendliness	Excellent: neighbors share cache lines	Poor when nodes are scattered

***Table 7.1**: releasing a list is harder: every node is deleted, one by one. Book: Sections 7.3.1 and 7.3.2.*

The Memory Each Element Costs

$$M_{\text{array}} = n\,d, \qquad M_{\text{list}} = n\,(d + p + a) \tag{7.5}$$

Equation (7.5): d data bytes, $p = 8$ pointer bytes, a allocator bytes per node.

An `int` node is padded to 16 bytes; glibc rounds it up to a 32-byte chunk — about 8 times the `int`.

For a 200-byte book record, the 8-byte pointer is a small tax.

> **THE ONE-SENTENCE SUMMARY.** An array is fast to *read* anywhere and slow to *change* in the middle; a linked list is fast to *change* anywhere you already hold a pointer and slow to *reach* anywhere else.

Cache lines are 64 bytes: an array uses every byte it loads, a scattered list loads a new line per node. Book: Sections 7.3.2 and 7.3.3.

Three Nodes by Hand

Listing 7.1 From `node_basics.cpp`: three nodes created with `new`, linked by hand and walked with `aux`.

```
    Node *a = new Node{13, nullptr};
    Node *b = new Node{3, nullptr};
    Node *c = new Node{86, nullptr};

    // Link them: a -> b -> c -> nullptr.
    a->next = b;
    b->next = c;
    Node *head = a;   // from now on the list is reached only through head

    std::cout << "Three nodes linked by hand: ";
    for (Node *aux = head; aux != nullptr; aux = aux->next) {
        std::cout << aux->data << " -> ";
    }
    std::cout << "NULL\n";
```

```
Three nodes linked by hand: 13 -> 3 -> 86 -> NULL
head->data             = 13
head->next->data       = 3
head->next->next->data = 86
```

***Listing 7.1**: `struct Node` is self-referential: it holds the address of a node, never a node. `head` is a pointer, not a node. The `for` loop over `aux` is the heartbeat of every list program; `head->next->next->data` is one arrow at a time. Book: Sections 7.4.1–7.4.3.*

`init`, `isEmpty` and `frontInsertion`

```
void init(TNode *&head) { head = nullptr; }
bool isEmpty(const TNode *head) { return head == nullptr; }
```

Pseudocode 7.1 `FRONT-INSERTION`: insert a new first node; two pointer assignments, in this order.

```
FRONT-INSERTION(L, x):
 1.  z ← a new node
 2.  z.data ← x
 3.  z.next ← L.head          // step 1: link the new node to the old first node
 4.  L.head ← z               // step 2: the new node becomes the first node
```

An uninitialized `head` does not mean "empty", it means "somewhere random": always `init`.

Pseudocode 7.1: link the new node to the old first node, *then* move `head`. With an empty list, the same two lines work.

Cost $O(1)$; n front insertions cost $\Theta(n)$, against $\Theta(n^2)$ for an array. Inserting 13 then 3 gives 3, 13: front insertion reverses the order, which is how a stack works. Book: Sections 7.5.1 and 7.5.2.

Front Insertion, Step by Step

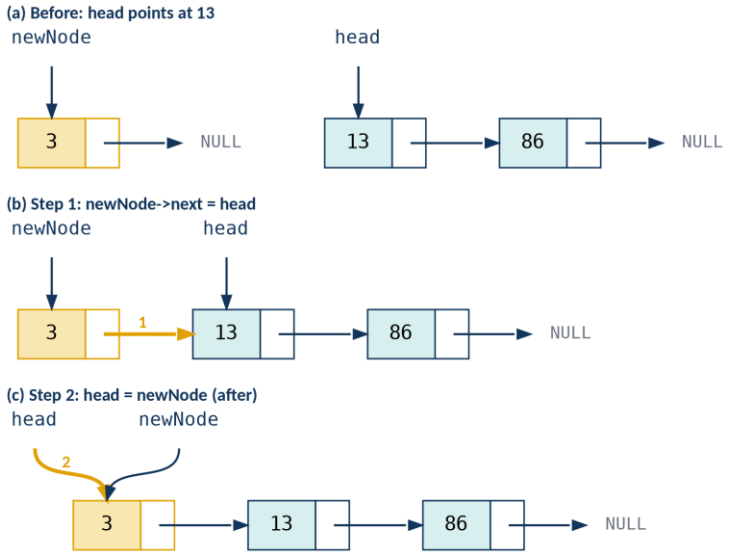


Figure 7.3 `frontInsertion(3)` on the list 13, 86: (a) before, (b) the new node is linked to the old first node, (c) `head` moves to the new node.

***Figure 7.3**: in panel (b) two pointers lead to 13. If line 4 ran before line 3, the new node would point at itself and 13 and 86 would be lost forever. Book: Section 7.5.2.*

`backInsertion`: A New Last Node

Pseudocode 7.2 `BACK-INSERTION` on a list with only a head pointer: walk to the last node, then attach.

```
BACK-INSERTION(L, x):
 1.  z ← a new node
 2.  z.data ← x
 3.  z.next ← NIL              // the new node will be the last node
 4.  if L.head = NIL           // empty list: the new node is also the first
 5.      L.head ← z
 6.  else
 7.      aux ← L.head
 8.      while aux.next ≠ NIL
 9.          aux ← aux.next
10.      aux.next ← z
```

The head-pointer list does not know where its last node is, so `aux` walks there.

The loop tests `aux.next ≠ NIL`: it must end *standing on* the last node.

The empty list needs its own branch: `aux->next` on a null `aux` is undefined behavior.

***Pseudocode 7.2**: $n - 1$ moves, $\Theta(n)$; building n values by back insertion is $\Theta(n^2)$. To change the link into a position, you must stand on the node before it. Book: Section 7.5.3.*

Back Insertion, Step by Step

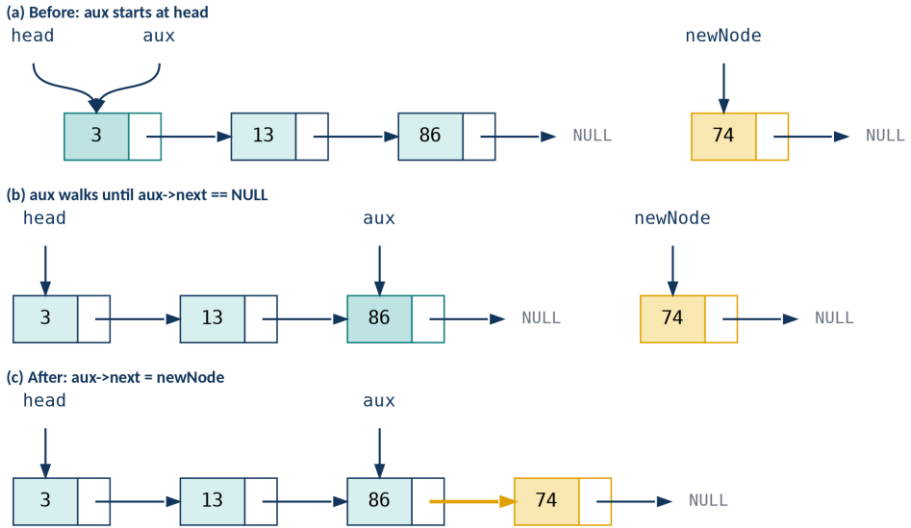


Figure 7.4 `backInsertion(74)` on the list 3, 13, 86: (a) `aux` starts at `head`, (b) `aux` stops at the node whose `next` is NULL, (c) that node's `next` becomes the new node.

Figure 7.4*: the loop test looks one node ahead. After two moves* `aux` *is on 86, whose* `next` *is NULL, so it stops there, not one step later; line 10 attaches 74. Book: Section 7.5.3.*

`show`: Walking the List

Pseudocode 7.3 `TRAVERSE`: visit every node from the first to the last.

```
TRAVERSE(L):
 1.  aux ← L.head
 2.  while aux ≠ NIL
 3.      visit aux            // e.g. print aux.data
 4.      aux ← aux.next
```

Here the test is `aux ≠ NIL`: we must visit the last node too.

How to choose: ask on which node the loop must *end*. Visited every node → test `aux`. Standing on the last node → test `aux.next`.

Choosing the wrong one is the most common off-by-one error in list code.

Pseudocode 7.3*: $\Theta(n)$, which is optimal — printing n values takes n steps. Book: Section 7.5.4.*

Traversal, Step by Step

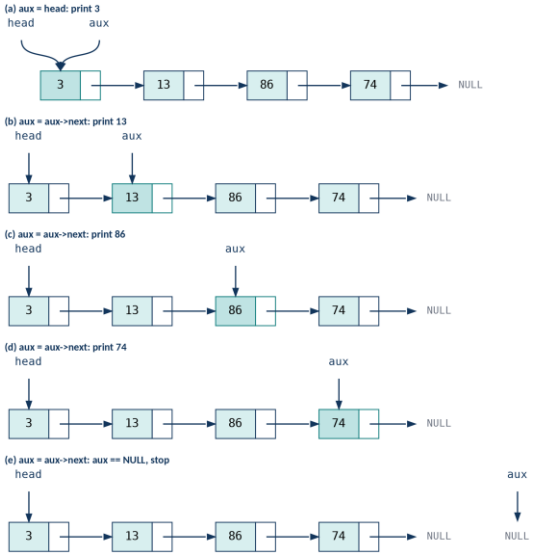


Figure 7.5 Traversing the list 3, 13, 86, 74: `aux` visits each node in turn and the loop ends when `aux` becomes NULL; `head` never moves.

> **THE RULE OF THE HEAD.** Never move `head` to walk through the list: every node it passes is lost. Walk with a helper pointer (`aux`, `cur`).

***Figure 7.5**: `aux` visits every node and ends on NULL; `head` never moves. Book: Sections 7.5 and 7.5.4.*

`frontDeletion`: Save, Move, Delete

Pseudocode 7.4 `FRONT-DELETION`: remember the first node, move `head` past it, then release it.

```
FRONT-DELETION(L):
 1.  if L.head = NIL
 2.      report "the list is empty" and return
 3.  del ← L.head              // step 1: remember the first node
 4.  L.head ← L.head.next      // step 2: the second node becomes the first
 5.  free del                  // step 3: give the old first node back
```

Deleting first, then `head = head->next`, reads a freed node.

Moving `head` first without saving the address leaks the node.

Save, move, delete — every deletion today follows it.

Pseudocode 7.4*: a one-node list becomes empty through line 4. Cost $O(1)$. Book: Section 7.5.5.*

Front Deletion, Step by Step

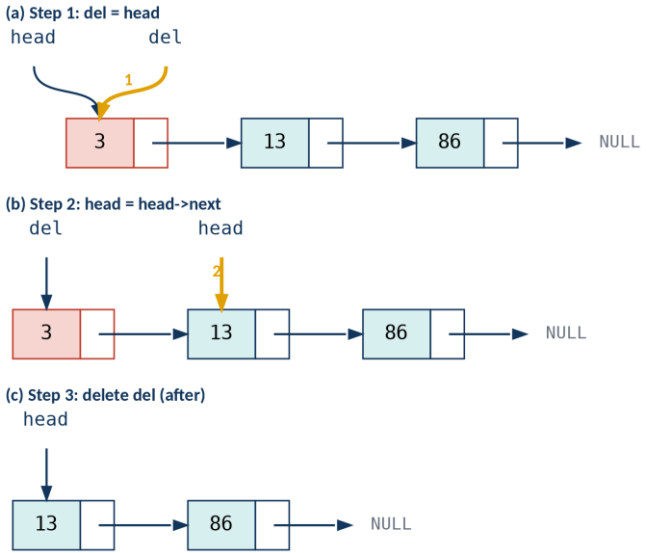


Figure 7.6 `frontDeletion()` on the list 3, 13, 86: (a) `del` remembers the first node, (b) `head` moves to the second node, (c) the old first node is deleted.

***Figure 7.6**: after step 2 the red node is no longer in the list, but it still exists and `del` still points at it; step 3 releases it. Book: Section 7.5.5.*

`backDeletion`: Removing the Last Node

Pseudocode 7.5 `BACK-DELETION` with only a head pointer: stop on the second-to-last node, cut the link, release the last node.

```
BACK-DELETION(L):
 1.  if L.head = NIL
 2.      report "the list is empty" and return
 3.  if L.head.next = NIL      // exactly one node
 4.      free L.head
 5.      L.head ← NIL
 6.  else
 7.      aux ← L.head
 8.      while aux.next.next ≠ NIL
 9.          aux ← aux.next
10.      del ← aux.next        // the last node
11.      aux.next ← NIL        // aux becomes the last node
12.      free del
```

`aux` must stop on the **second-to-last** node, where `aux.next.next` is NIL.

One node is a separate case: `aux.next.next` would dereference NULL, and it is `head` that must change.

Line 11 is the line most often forgotten: without it, 86 still points at the freed node.

Pseudocode 7.5*: $n - 2$ moves, $\Theta(n)$ — and a tail pointer will not fix it. Book: Section 7.5.6.*

Back Deletion, Step by Step

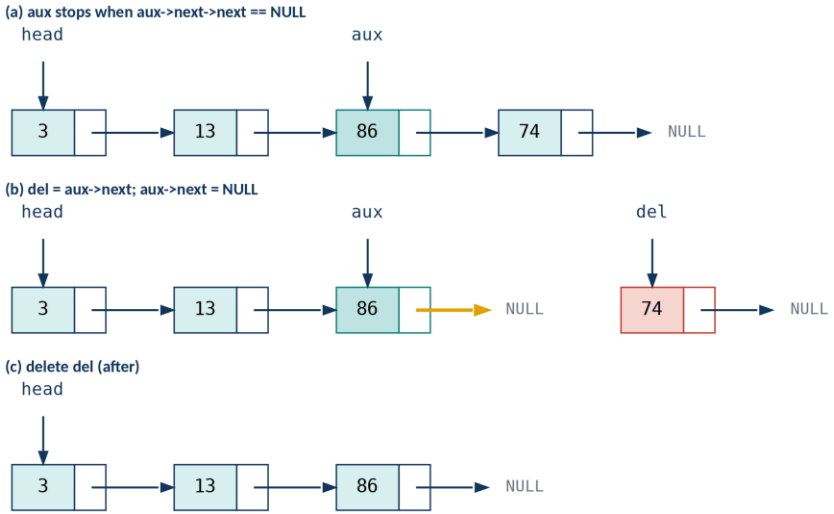


Figure 7.7 `backDeletion()` on the list 3, 13, 86, 74: (a) `aux` stops on the second-to-last node, (b) the last node is detached, (c) it is deleted.

***Figure 7.7**: (a) `aux` stops on 86 because the node after it has a NULL `next`; (b) the link is cut and 74 is remembered in `del`; (c) it is released. Book: Section 7.5.6.*

`clear` and the Costs So Far

Table 7.2 Running time of each operation of the head-pointer list with n nodes.

Operation	Cost	Why
`init`, `isEmpty`	$O(1)$	one assignment or comparison
`frontInsertion`	$O(1)$	two pointer assignments (**Pseudocode 7.1**)
`backInsertion`	$\Theta(n)$	walk to the last node (**Pseudocode 7.2**)
`show` (traverse)	$\Theta(n)$	visit every node (**Pseudocode 7.3**)
`frontDeletion`	$O(1)$	save, move `head`, delete (**Pseudocode 7.4**)
`backDeletion`	$\Theta(n)$	walk to the second-to-last node (**Pseudocode 7.5**)
`clear`	$\Theta(n)$	delete every node

***Table 7.2**: `clear` is repeated front deletion, $\Theta(n)$, and leaves a valid empty list. Call it before the last pointer to the list disappears, or every node leaks. Book: Sections 7.5.7 and 7.5.8.*

The Complete Program: Back Deletion

Listing 7.2 From `sll_head.cpp`: `backDeletion`, with the one-node case and the cut link.

```
void backDeletion(TNode *&head) {
    if (isEmpty(head)) {
        std::printf("The linked list is empty\n");
        return;
    }
    int d;
    if (head->next == nullptr) {          // exactly one node
        d = head->data;
        delete head;
        head = nullptr;
    } else {
        TNode *aux = head;
        while (aux->next->next != nullptr) {   // stop at the second-to-last node
            aux = aux->next;
        }
        TNode *del = aux->next;
        d = del->data;
        aux->next = nullptr;              // aux becomes the new last node
        delete del;
    }
    std::printf("%d has been deleted by backDeletion()\n", d);
}
```

***Listing 7.2**: line 83 is line 11 of **Pseudocode 7.5**. The run leaves 13, 86. Book: Section 7.5.8.*

The Same List in Java and Python

Listing 7.3 From `SllHead.java`: front deletion without `delete`.

```java
    void frontDeletion() {
        if (isEmpty()) {
            System.out.println("The linked list is empty");
            return;
        }
        int d = head.data;
        head = head.next;                // the old first node becomes garbage
        System.out.println(d + " has been deleted by frontDeletion()");
    }
```

Listing 7.4 From `sll_head.py`: the same in Python.

```python
    def front_deletion(self):
        if self.is_empty():
            print("The linked list is empty")
            return
        d = self.head.data
        self.head = self.head.next            # the old first node becomes garbage
        print(f"{d} has been deleted by frontDeletion()")
```

Listing 7.3, ***Listing 7.4***: *the old first node becomes unreachable and the garbage collector reclaims it;* `clear` *is just* `head = null`*. No use after free, but you can still lose nodes (moving* `head`*) or keep them alive by a forgotten reference.* `__slots__` *saves memory and catches typos. Book: Section 7.5.9.*

`search` and `insertAfter`

Pseudocode 7.6 `SEARCH`: return the first node whose data equals `key`, or NIL.

```
SEARCH(L, key):
 1.  aux ← L.head
 2.  while aux ≠ NIL and aux.data ≠ key
 3.      aux ← aux.next
 4.  return aux                // the node holding key, or NIL if there is none
```

Pseudocode 7.7 `INSERT-AFTER`: insert a new node right after node `p` in $O(1)$ time.

```
INSERT-AFTER(p, x):           // p is a node of the list, not NIL
 1.  z ← a new node
 2.  z.data ← x
 3.  z.next ← p.next          // step 1: the new node points at p's successor
 4.  p.next ← z               // step 2: p points at the new node
```

***Pseudocode 7.6**: `aux ≠ NIL` comes first (short-circuit), and the node is returned, ready for **Pseudocode 7.7**. $\Theta(n)$ worst case; no $O(\lg n)$ binary search, since reaching the middle costs $n/2$ steps (Equation (7.2)). `insertAfter` is $O(1)$ only because the caller already holds `p`. Book: Sections 7.6.1 and 7.6.3.*

`length`, Iteratively and Recursively

DEFINITION (recursive). A list is either *empty* (the pointer is NULL), or a *first node* followed by a *list* (the one that starts at the first node's `next`).

$$L(p) = \begin{cases} 0 & \text{if } p = \text{NULL} \\ 1 + L(p.\text{next}) & \text{otherwise} \end{cases} \tag{7.6}$$

```
int lengthRec(const Node *head) {
    if (head == nullptr) return 0;          // the empty list has length 0
    return 1 + lengthRec(head->next);       // one node plus the rest
}
```

Equation (7.6): $L(20) = 1 + (1 + (1 + 0)) = 3$. Both versions take $\Theta(n)$ time, but the recursion keeps $n + 1$ calls on the call stack (8 MB on Linux): a list of millions can overflow it. Book: Section 7.6.2.

Insertion in the Middle, Step by Step

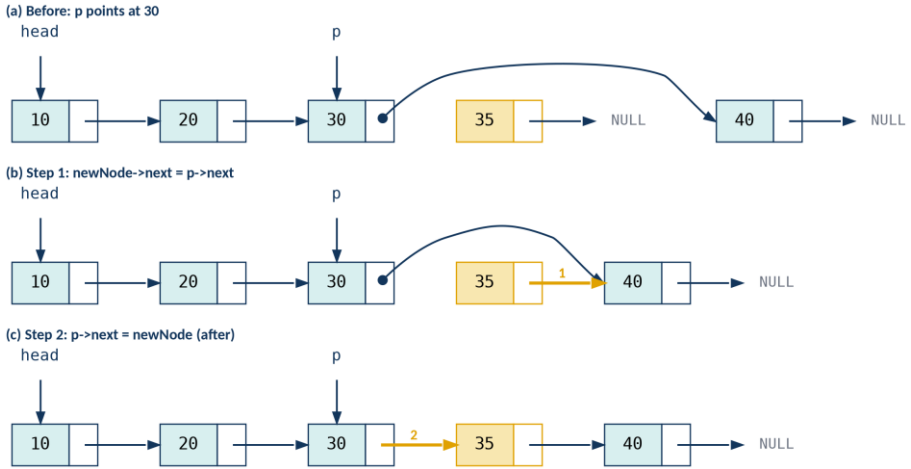


Figure 7.8 `insertAfter(p, 35)` with `p` at 30: (a) before, (b) the new node points at `p`'s successor, (c) `p` points at the new node.

Figure 7.8*: in panel (b) the new node already points at 40 while 30 still does too; nothing is broken until step 2 links it in. Reversing the steps makes the new node point at itself. Inserting before a node needs a lagging pointer (Exercise 7.9). Book: Section 7.6.3.*

`insertSorted`: Keeping the List in Order

Pseudocode 7.8 `INSERT-SORTED`: insert `x` so that the list stays in ascending order.

```
INSERT-SORTED(L, x):
 1.  if L.head = NIL or x < L.head.data
 2.      FRONT-INSERTION(L, x)
 3.      return
 4.  aux ← L.head
 5.  while aux.next ≠ NIL and aux.next.data < x
 6.      aux ← aux.next
 7.  INSERT-AFTER(aux, x)
```

`aux` walks while the *next* node's value is smaller; it stops on the last smaller node.

Special case (line 1): an empty list, or a value smaller than the first, changes `head`.

`<` puts equal keys after existing ones: the insertion is **stable**.

***Pseudocode 7.8**: $\Theta(n)$ worst case. Inserting n values this way is insertion sort on a list — $\Theta(n^2)$, but without a single element shifted. Book: Section 7.6.4.*

Sorted Insertion, Both Cases

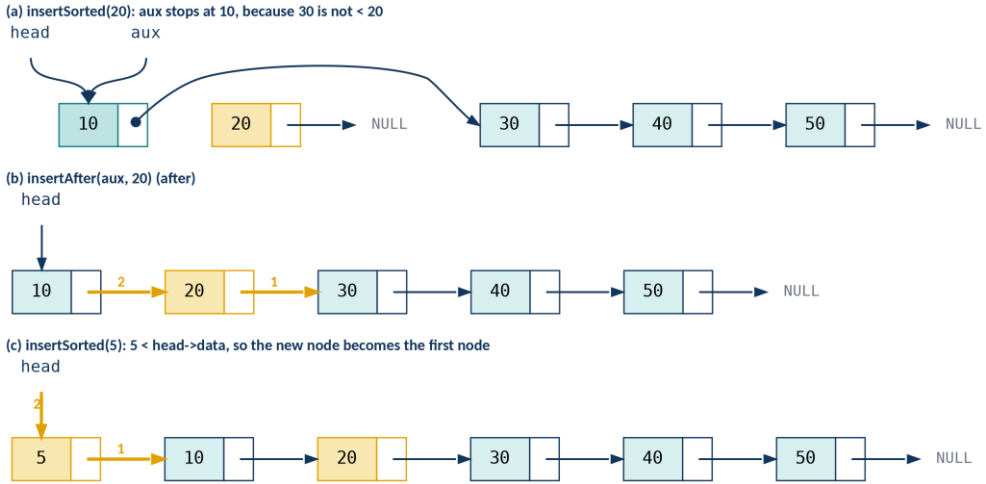


Figure 7.9 `insertSorted`: (a) for 20, `aux` stops on the last node smaller than 20, (b) the new node is inserted after it, (c) 5 is smaller than every value and becomes the first node.

***Figure 7.9**: for 20, `aux` stops on 10 because the next node holds 30; 5 is smaller than every value and becomes the first node. Book: Section 7.6.4.*

`deleteKey`: Deletion by Value

Pseudocode 7.9 `DELETE-KEY`: remove the first node whose data equals `key`, using a trailing pointer `prev`.

```
DELETE-KEY(L, key):
 1.  prev ← NIL
 2.  cur ← L.head
 3.  while cur ≠ NIL and cur.data ≠ key
 4.      prev ← cur
 5.      cur ← cur.next
 6.  if cur = NIL
 7.      return FALSE           // key is not in the list
 8.  if prev = NIL              // cur is the first node
 9.      L.head ← cur.next
10.  else
11.      prev.next ← cur.next   // bypass cur
12.  free cur
13.  return TRUE
```

First node: `prev` stays NIL, `head` moves. **Middle:** line 11 bypasses `cur`.

Last node: `prev.next` becomes NIL — back deletion with no extra case. **Absent:** nothing changes.

Pseudocode 7.9*: two pointers walk together,* `prev` *one node behind* `cur`*. The empty list is the absent case with a loop that never starts.* $\Theta(n)$ *worst case. Book: Section 7.6.5.*

Deletion by Key, Step by Step

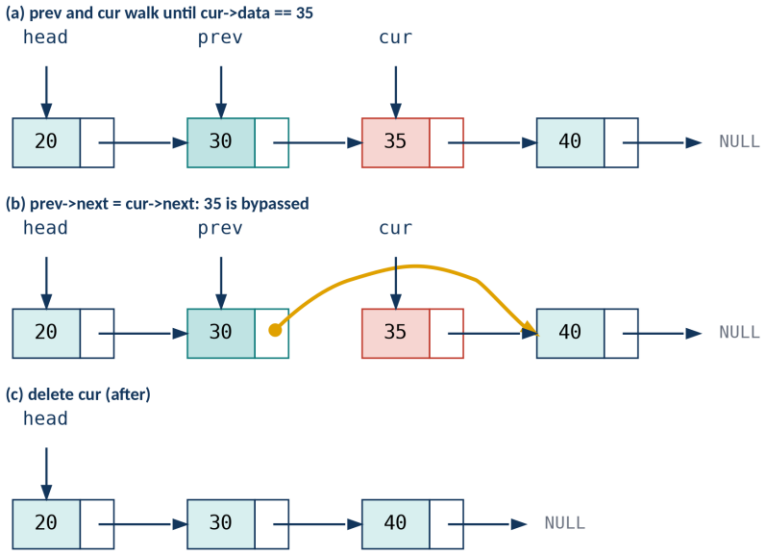


Figure 7.10 `deleteKey(35)`: (a) `prev` and `cur` stop with `cur` on 35, (b) `prev->next` skips over it, (c) the node is deleted.

Figure 7.10: *`cur` on 35, `prev` on 30; `prev->next` skips 35, so a walk from `head` no longer meets it; then it is released. Book: Section 7.6.5.*

Reversing a List Iteratively

Pseudocode 7.10 `REVERSE-ITERATIVE`: reverse the list in one pass by turning each `next` arrow around.

```
REVERSE-ITERATIVE(L):
 1.  prev ← NIL
 2.  cur ← L.head
 3.  while cur ≠ NIL
 4.      nxt ← cur.next       // 1. remember the rest of the list
 5.      cur.next ← prev      // 2. turn this node's arrow around
 6.      prev ← cur           // 3. step both pointers forward
 7.      cur ← nxt
 8.  L.head ← prev
```

LOOP INVARIANT. At the start of each iteration, `prev` is the head of a correctly reversed list of the nodes already visited, and `cur` is the head of the untouched list of the nodes not yet visited; together they contain every node exactly once.

Pseudocode 7.10*: line 4 saves the only record of the rest before line 5 overwrites it. When* `cur` *is NIL,* `prev` *heads the whole reversed list.* $\Theta(n)$ *time,* $O(1)$ *memory. Book: Section 7.6.6.*

Iterative Reversal, Traced

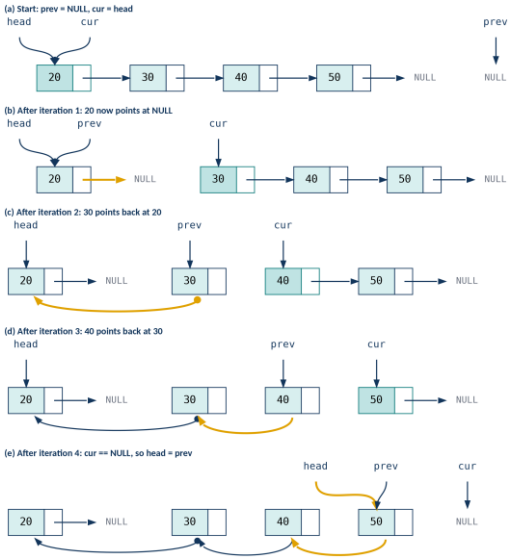


Figure 7.11 Iterative reversal of 20, 30, 40, 50: after each iteration one more arrow points backward; when `cur` reaches NULL, `prev` is the new first node.

***Figure 7.11**: after each iteration one more arrow points backward. No data moves. Book: Section 7.6.6.*

Reversing a List Recursively

Pseudocode 7.11 `REVERSE-RECURSIVE`: reverse the rest of the list, then hang the first node on its end.

```
REVERSE-RECURSIVE(head):      // returns the head of the reversed list
 1.  if head = NIL or head.next = NIL
 2.      return head
 3.  newHead ← REVERSE-RECURSIVE(head.next)
 4.  head.next.next ← head    // the old second node now points back at head
 5.  head.next ← NIL          // head becomes the last node
 6.  return newHead
```

$$R(n) = R(n-1) + c, \quad R(1) = c \quad \Rightarrow \quad R(n) = c\,n = \Theta(n) \tag{7.7}$$

Pseudocode 7.11*: after line 3,* `head.next` *still points at the old second node, now the last of the reversed rest, so* `head.next.next ← head` *needs no walk. Equation (7.7):* $\Theta(n)$ *time, but* $\Theta(n)$ *stack. "Solve the rest, then fix up the first node" returns in Chapters 10 and 11. Book: Section 7.6.7.*

Recursive Reversal, Traced

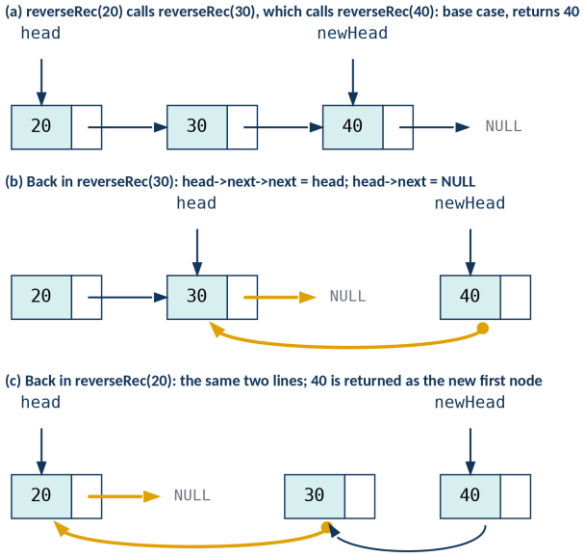


Figure 7.12 Recursive reversal of 20, 30, 40: the deepest call returns 40; each returning call turns one arrow around and cuts its own node's forward link.

Figure 7.12*: the deepest call returns 40; each returning call turns one arrow around and cuts its own node's forward link. Book: Section 7.6.7.*

Costs of the New Operations

Table 7.3 Cost of the operations of Section 7.6 on a list of n nodes.

Operation	Time	Extra memory
`search`	$\Theta(n)$ worst, $O(1)$ best	$O(1)$
`lengthIter` / `lengthRec`	$\Theta(n)$ / $\Theta(n)$	$O(1)$ / $\Theta(n)$ call stack
`insertAfter(p, x)`	$O(1)$	$O(1)$
`insertSorted`	$\Theta(n)$ worst	$O(1)$
`deleteKey`	$\Theta(n)$ worst	$O(1)$
`reverseIter` / `reverseRec`	$\Theta(n)$ / $\Theta(n)$	$O(1)$ / $\Theta(n)$ call stack
`destroy`	$\Theta(n)$	$O(1)$

***Table 7.3**: every walk is $\Theta(n)$; only `insertAfter` with a known `p` is $O(1)$. The recursive versions pay $\Theta(n)$ in call stack. Book: Section 7.6.9.*

A Dummy Head Node

DEFINITION. A singly linked list *with a dummy head node* always begins with one extra node whose data is meaningless; the first real element is `dummy->next`, and the list is empty exactly when `dummy->next` is NULL.

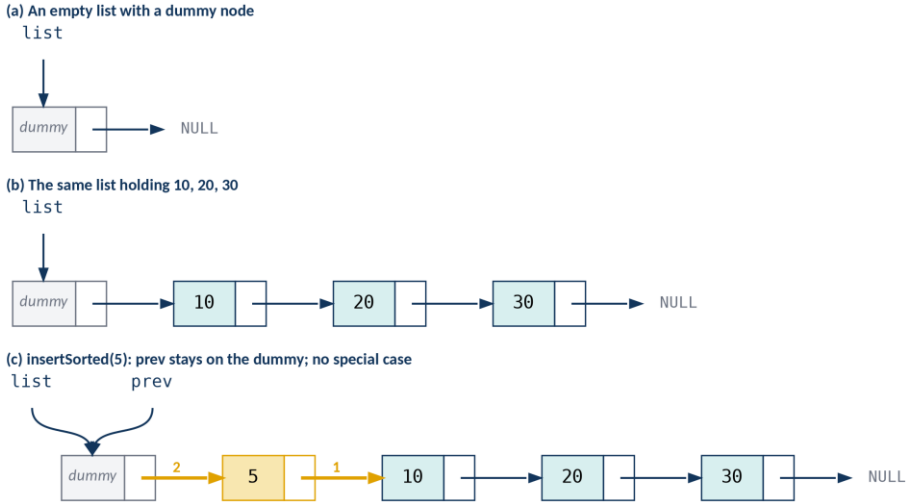


Figure 7.13 A list with a dummy head node: (a) the empty list is one dummy node, (b) a list holding 10, 20, 30, (c) inserting 5 before 10 is an insertion after the dummy, with no special case.

***Figure 7.13**: every special case so far came from the first node being pointed at by `head`, not by a `next` field. With a dummy, inserting 5 before 10 is an insertion after the dummy. Book: Section 7.7.1.*

Operations Without Special Cases

Listing 7.5 From `sll_dummy.cpp`: sorted insertion and deletion by key, each one loop and one link change.

```
void insertSorted(Node *dummy, int x) {      // one loop, no "new first node" branch
    Node *prev = dummy;
    while (prev->next != nullptr && prev->next->data < x) prev = prev->next;
    prev->next = new Node{x, prev->next};
}

bool deleteKey(Node *dummy, int key) {       // prev always exists: it may be the dummy
    Node *prev = dummy;
    while (prev->next != nullptr && prev->next->data != key) prev = prev->next;
    if (prev->next == nullptr) return false;
    Node *del = prev->next;
    prev->next = del->next;
    delete del;
    return true;
}
```

```
insertSorted x3:    [dummy] -> 10 -> 20 -> 30 -> NULL
front 5, back 99:   [dummy] -> 5 -> 10 -> 20 -> 30 -> 99 -> NULL
delete 5, 20, 99:   [dummy] -> 10 -> 30 -> NULL
deleteFront x2:     [dummy] -> NULL
```

***Listing 7.5**: deleting 5, the first real element, runs exactly the same code as deleting 20 or 99. Traversals start at `dummy->next`; `destroy` frees the dummy too. Price: one node per list. Book: Section 7.7.2.*

A Third Way: A Pointer to the Link

Listing 7.6 From `sll_link.cpp`: `Node **link` starts as `&head` and becomes `&node->next`.

```
void insertSorted(Node **link, int x) {
    while (*link != nullptr && (*link)->data < x) link = &(*link)->next;
    *link = new Node{x, *link};          // rewrites head OR some node's next
}

bool deleteKey(Node **link, int key) {
    while (*link != nullptr && (*link)->data != key) link = &(*link)->next;
    if (*link == nullptr) return false;
    Node *del = *link;
    *link = del->next;                   // the link now skips the deleted node
    delete del;
    return true;
}
```

```
sorted:          5 -> 10 -> 20 -> 20 -> 20 -> 30 -> NULL
deleteKey(5):    10 -> 20 -> 20 -> 20 -> 30 -> NULL
```

***Listing 7.6**: `*link = ...` changes `head` or a `next` field with the same line. No equivalent in Java or Python; there the dummy is the tool. Which variant? Head pointer: simplest. Dummy: shortest code, one node extra. Pointer to the link: short code, C and C++ only. Book: Sections 7.7.3 and 7.7.4.*

Head and Tail Pointers

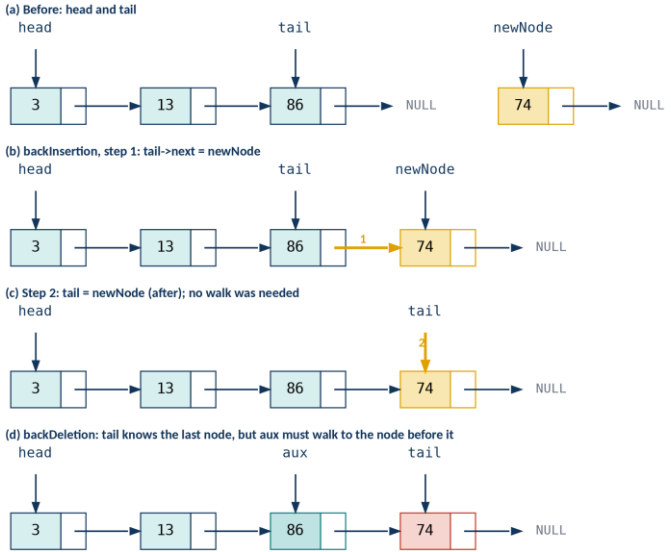


Figure 7.14 A list with `head` and `tail`: (a) before `backInsertion(74)`, (b) the old last node points at the new node, (c) `tail` moves to it; (d) `backDeletion` still has to walk `aux` to the node before `tail`.

***Figure 7.14**: back insertion attaches to `tail` and moves it: $O(1)$. Moving between "empty" and "one node" must change both pointers. Book: Section 7.8.1.*

Why `tail` Cannot Speed Up Back Deletion

> **THE HEAD-TAIL INVARIANT.** Either the list is empty and `head` and `tail` are both NULL, or `head` points at the first node, `tail` at the last, and `tail->next` is NULL.

`tail` says *which* node to delete, but the node **before** it must become the new last node.

A singly linked node records who comes after it, never who comes before it.

So we walk from `head` to the node whose `next` is `tail`: $n - 2$ steps, $\Theta(n)$.

A pointer to the second-to-last node helps only once. The real fix: a link to the predecessor in every node — Chapter 9.

*Panel (d) of **Figure 7.14** shows the walk. Book: Section 7.8.2.*

The Variants Compared

Table 7.4 The singly linked list variants compared, for a list of n nodes.

Operation	Head only	Head and tail	Dummy head node
Insert at the front	$O(1)$	$O(1)$	$O(1)$
Insert at the back	$\Theta(n)$	$O(1)$	$\Theta(n)$ (or $O(1)$ with a tail)
Delete at the front	$O(1)$	$O(1)$	$O(1)$
Delete at the back	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Search, delete by key	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Special cases for the first node	yes	yes, and for `tail`	none
Extra memory per list	1 pointer	2 pointers	1 pointer and 1 node

```
3 has been deleted by frontDeletion()
61 has been deleted by backDeletion() after 3 step(s)
52 has been deleted by backDeletion() after 2 step(s)
74 has been deleted by backDeletion() after 1 step(s)
```

***Table 7.4**: only back deletion resists every variant. The run of `sll_head_tail.cpp`: on lists of 5, 4 and 3 nodes the walks take 3, 2 and 1 steps, exactly $n - 2$. Book: Section 7.8.3.*

The Circular Singly Linked List

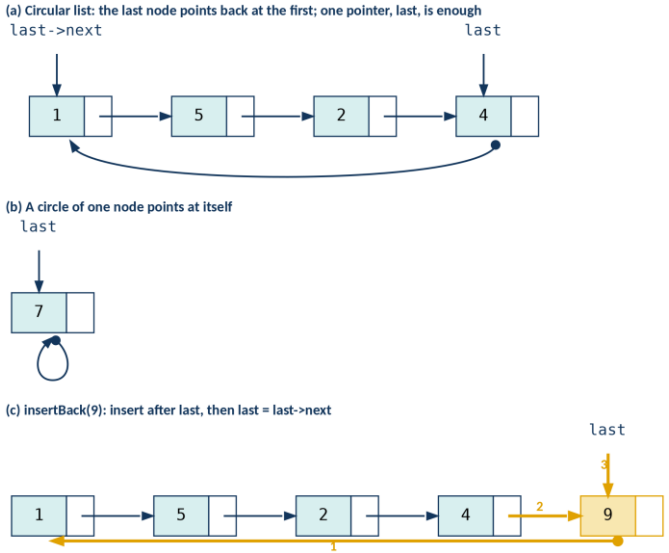


Figure 7.15 A circular singly linked list: (a) four nodes, reached through one pointer `last`, with the first node at `last->next`; (b) a one-node circle points at itself; (c) inserting at the back is an insertion after `last` followed by moving `last`.

Figure 7.15*: one pointer,* `last`*, reaches both ends, since the first node is* `last->next`*. Front and back insertion and front deletion are* $O(1)$*; back deletion is still* $\Theta(n)$*. A one-node circle points at itself. Book: Section 7.9.1.*

Traversal Needs a New Stop Signal

```
  const Node *aux = last->next;          // start at the first node
  do {
      visit(aux);
      aux = aux->next;
  } while (aux != last->next);           // stop when we come back to the start
```

Listing 7.7 From `csll.cpp`: `deleteAfter` keeps `last` correct.

```
void deleteAfter(Node *&last, Node *prev) {
    Node *del = prev->next;
    if (del == prev) {                   // the only node
        last = nullptr;
    } else {
        prev->next = del->next;
        if (del == last) last = prev;    // deleting the last node moves last back
    }
    delete del;
}
```

*`while (aux != nullptr)` would print the ring forever; `while (aux != last->next)` would stop before visiting anything. The `do-while` tests after the body. **Listing 7.7**: `del == prev` detects the only node. Book: Sections 7.9.1 and 7.9.2.*

The Session, Minute by Minute

```
Queue for the one reading desk: Dewi(1) Ani(5) Budi(2) Citra(4)
t= 1  Dewi  read 1 min and is done
t= 3  Ani   read 2 min, 3 min left
t= 5  Budi  read 2 min and is done
t= 7  Citra read 2 min, 2 min left
t= 9  Ani   read 2 min, 1 min left
t=11  Citra read 2 min and is done
t=12  Ani   read 1 min and is done
After the session: (empty)
```

Dewi (1 min, priority pass) leaves at $t = 1$; Budi leaves at $t = 5$.

Ani and Citra continue around the shrinking ring with no “end of list” code.

Total 12 minutes $= 1 + 5 + 2 + 4$: no time wasted.

`prev` stands on the node before the reader being served, as `deleteAfter` needs: round-robin scheduling with a 2-minute time slice. Josephus (Exercise 7.11) and cycle detection (Exercise 7.6) are two more circular classics. Book: Section 7.9.3.

Stacks and Queues on a Linked List

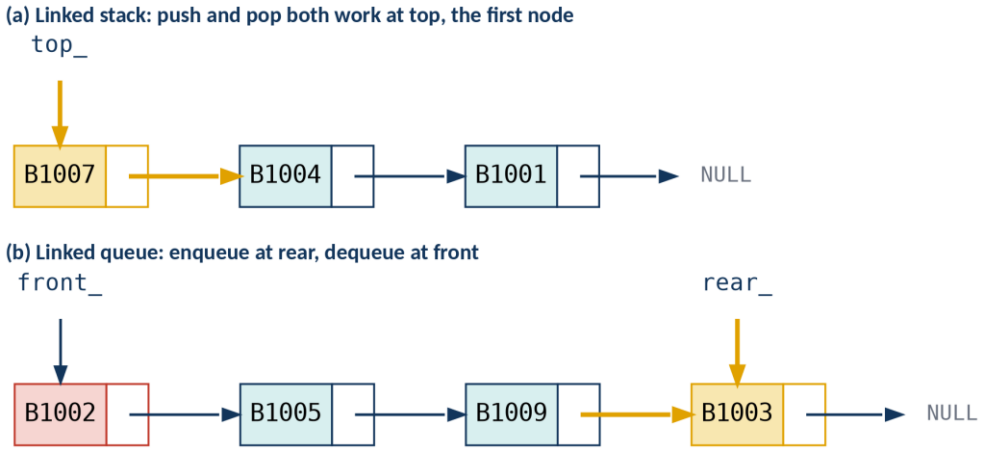


Figure 7.16 (a) A linked stack: `push` and `pop` work at the first node, `top_`. (b) A linked queue: `enqueue` links a new node after `rear_`, `dequeue` removes the node at `front_`.

Stack: `push` = front insertion, `pop` = front deletion, both $O(1)$; one pointer, `top_`.

Queue: `enqueue` at the tail (`rear_`), `dequeue` at the head (`front_`), both $O(1)$. The other direction would make `dequeue` a $\Theta(n)$ back deletion.

Figure 7.16*: no capacity and no index arithmetic, unlike Chapter 5. Book: Section 7.10.1.*

The Linked Queue

Listing 7.8 From `linked_stack_queue.cpp`: `dequeue` for the Pustaka Ceria hold queue.

```
    Book dequeue() {                        // front deletion: O(1)
        if (isEmpty()) throw std::underflow_error("dequeue from an empty queue");
        Node *del = front_;
        Book b = del->data;
        front_ = front_->next;
        if (front_ == nullptr) rear_ = nullptr;   // the queue became empty
        delete del;
        --size_;
        return b;
    }
```

```
Hold requests waiting: 3
  serve Negeri 5 Menara
  serve Ronggeng Dukuh Paruk
  serve Cantik Itu Luka
  serve Pulang
Error caught: dequeue from an empty queue
```

***Listing 7.8**: `if (front_ == nullptr) rear_ = nullptr;` — without it, `rear_` points at a freed node, and a version that tests `rear_` would write into freed memory (Exercise 7.12). Book: Section 7.10.2.*

Three Safety Features of the Classes

Listing 7.9 From `linked_stack_queue.cpp`: the node type is private, copying is forbidden, the destructor frees every node.

```
class LinkedStack {                        // top of the stack = first node
    struct Node { Book data; Node *next; };
    Node *top_ = nullptr;
    int size_ = 0;
public:
    LinkedStack() = default;
    LinkedStack(const LinkedStack &) = delete;              // no accidental copies
    LinkedStack &operator=(const LinkedStack &) = delete;
    ~LinkedStack() { while (!isEmpty()) pop(); }            // frees every node
```

Private node: no outside code can break the links.

Destructor (RAII): the nodes left in the structures at the end of `main` are released automatically.

No copies (rule of three): a default copy would share `top_`, and both destructors would delete each node.

Listing 7.9*: Java and Python need no destructors. Linked or array-based? Arrays use less memory and are faster in practice;* `std::stack` *uses* `std::deque`*. Linked wins for large elements or when a capacity is unacceptable. Book: Sections 7.10.2 and 7.10.3.*

Four Memory Bugs, One Order Mistake

Memory leak: a node is never deleted. **Use after free:** a node is used after `delete`.

Dangling link: a `next` or `tail` still points at a freed node. **Double free:** the same node deleted twice.

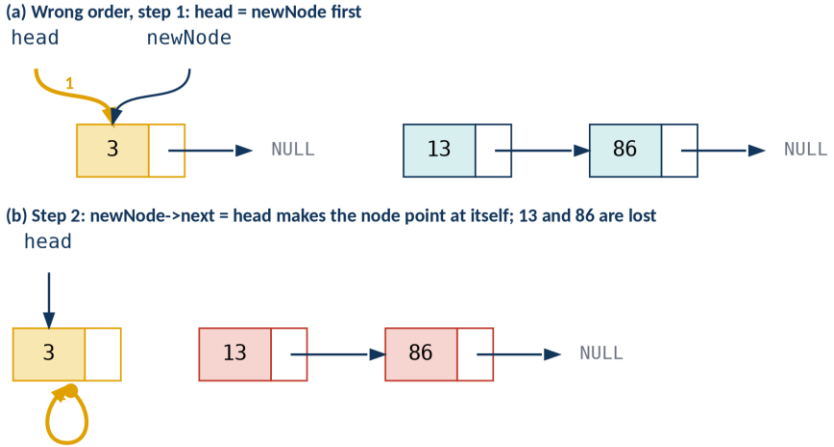


Figure 7.17 Front insertion with its two assignments in the wrong order: (a) `head` moves first, (b) the new node then points at itself and the old nodes 13 and 86 can never be reached again.

***Figure 7.17**: `head` moves first, then the new node points at itself: the list reads 3, 3, 3 forever and 13 and 86 leak. These bugs usually produce no visible symptom. Book: Section 7.11.1.*

Valgrind: A Leak

```
==1776== HEAP SUMMARY:
==1776==     in use at exit: 48 bytes in 3 blocks
==1776==   total heap usage: 5 allocs, 2 frees, 77,872 bytes allocated
==1776== 48 (16 direct, 32 indirect) bytes in 1 blocks are definitely lost in loss record 2 of 2
==1776==    by 0x10920F: build() (memory_bugs.cpp:17)
==1776==    by 0x109369: main (memory_bugs.cpp:42)
==1776== LEAK SUMMARY:
==1776==    definitely lost: 16 bytes in 1 blocks
==1776==    indirectly lost: 32 bytes in 2 blocks
```

Read the report **from the bottom up**: 16 bytes *definitely* lost, 32 *indirectly* lost.

Only the first node was pointed at by `head`; the other two are reachable only from a lost node.

Allocated by `build()` at line 17, called from `main` at line 42. Fix the definite leak and the rest follows.

`memory_bugs.cpp` switches on one bug per argument: `leak`, `uaf`, `dangle`, `double`. Compile with `-g -O0`; Valgrind runs 20 to 50 times more slowly. Book: Sections 7.11.1 and 7.11.2.

Valgrind: A Use After Free

```
==1786== Invalid read of size 4
==1786==    at 0x109247: show(Node const*) (memory_bugs.cpp:22)
==1786==    by 0x1093DB: main (memory_bugs.cpp:49)
==1786==  Address 0x4e260d0 is 0 bytes inside a block of size 16 free'd
==1786==    by 0x1092C3: clearRight(Node*&) (memory_bugs.cpp:30)
==1786==    by 0x1093CF: main (memory_bugs.cpp:48)
==1786==  Block was alloc'd at
==1786==    by 0x10920F: build() (memory_bugs.cpp:17)
==1786==    by 0x109369: main (memory_bugs.cpp:42)
1 2 3
2 3
```

The program printed `2 3`, the right values, from nodes already deleted. Three traces: the invalid read (`show`, line 22), where the block was freed (`clearRight`, line 30), and where it was allocated (`build`, line 17). With `-O2` and no Valgrind it crashed. Book: Section 7.11.2.

The Certificate, and AddressSanitizer

```
==3636== HEAP SUMMARY:
==3636==     in use at exit: 0 bytes in 0 blocks
==3636==   total heap usage: 6 allocs, 6 frees, 77,888 bytes allocated
==3636==
==3636== All heap blocks were freed -- no leaks are possible
==3636==
==3636== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

`All heap blocks were freed` and `ERROR SUMMARY: 0 errors`: every C++ program of the chapter produced both lines.

AddressSanitizer: the compiler inserts checks and poisons *shadow memory* around each block; about twice as slow; Linux, macOS and Windows.

`-fsanitize=address,undefined`; LeakSanitizer reports leaks at exit.

Book: Sections 7.11.2 and 7.11.3.

ASan: The Dangling Link

```
READ of size 4 at 0x502000000010 thread T0
    #0 0x55d758b734df in show(Node const*) memory_bugs.cpp:22
    #1 0x55d758b73cac in main memory_bugs.cpp:52
0x502000000010 is located 0 bytes inside of 16-byte region [0x502000000010,0x502000000020)
freed by thread T0 here:
    #1 0x55d758b73831 in backDeletionWrong(Node*) memory_bugs.cpp:37
    #2 0x55d758b73c8d in main memory_bugs.cpp:51
previously allocated by thread T0 here:
    #1 0x55d758b733d7 in build() memory_bugs.cpp:17
    #2 0x55d758b739ba in main memory_bugs.cpp:42
SUMMARY: AddressSanitizer: heap-use-after-free memory_bugs.cpp:22 in show(Node const*)
```

`backDeletionWrong` deleted the last node at line 37 but left the link; `show` read `data` at offset 0 of the 16-byte node. For `double`, ASan stops at the read of `next` (`8 bytes inside`) before the second `delete`; LeakSanitizer's "direct" and "indirect" match Valgrind's "definitely" and "indirectly". Book: Section 7.11.3.

A Checklist for Every List Function

1. **Save, then move, then delete.**
2. **Leave no pointer at a freed node:** `head`, `tail`, a predecessor's `next`, a saved helper.
3. **Every `new` has exactly one owner that deletes it;** copying either copies the nodes or is forbidden.
4. **Check the four shapes:** empty, one node, and the first, a middle and the last node.
5. **Run the tools** every time: ASan or `valgrind --leak-check=full`.

A test suite that has never run under a memory checker proves much less than it seems to. Book: Section 7.11.4.

C++: `std::forward_list` and `std::list`

Listing 7.10 From `std_lists.cpp`: the singly linked `std::forward_list`.

```
    std::forward_list<int> f = {3, 1, 4, 1, 5};
    print("start:                ", f);
    f.push_front(9);                          // O(1): a new first node
    print("add 9 at the front:   ", f);
    f.insert_after(f.begin(), 2);             // O(1): after a known node
    print("insert 2 after first:", f);
    f.remove(1);                              // O(n): every node holding 1
    print("remove every 1:       ", f);
    f.reverse();                              // O(n): relinks, copies nothing
    print("reverse:              ", f);
```

```
start:                3 1 4 1 5
add 9 at the front:   9 3 1 4 1 5
insert 2 after first: 9 2 3 1 4 1 5
remove every 1:       9 2 3 4 5
reverse:              5 4 3 2 9
```

***Listing 7.10**: `insert_after`, `before_begin()` (the dummy idea), no `push_back`, no `size()`. `std::list` is doubly linked. Book: Section 7.12.1.*

The Libraries Compared

Table 7.5 The linked lists of the C++, Java and Python standard libraries.

	C++ `std::forward_list`	C++ `std::list`	Java `LinkedList`	Python `collections.deque`
Structure	singly linked	doubly linked	doubly linked	doubly linked list of 64-slot blocks
Front insert / delete	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Back insert / delete	not provided	$O(1)$	$O(1)$	$O(1)$
Insert after a known position	$O(1)$ `insert_after`	$O(1)$ `insert`	$O(1)$ via `ListIterator.add`	$O(n)$ `insert`
Access by index	not provided	not provided	$\Theta(n)$ `get(i)`	$O(1)$ at the ends, $O(n)$ in the middle
Size	$\Theta(n)$ walk	$O(1)$	$O(1)$	$O(1)$
Sort in place	`sort()`, merge sort	`sort()`, merge sort	via `Collections.sort` (array copy)	not provided

***Table 7.5**: Java's `LinkedList` is doubly linked; `ArrayList` and `ArrayDeque` usually win. Python's `list` is a dynamic array; `deque` is a doubly linked list of 64-slot blocks. A hand-written Python node costs about 48 bytes. Book: Sections 7.12.1–7.12.3.*

Measuring: Array Against List

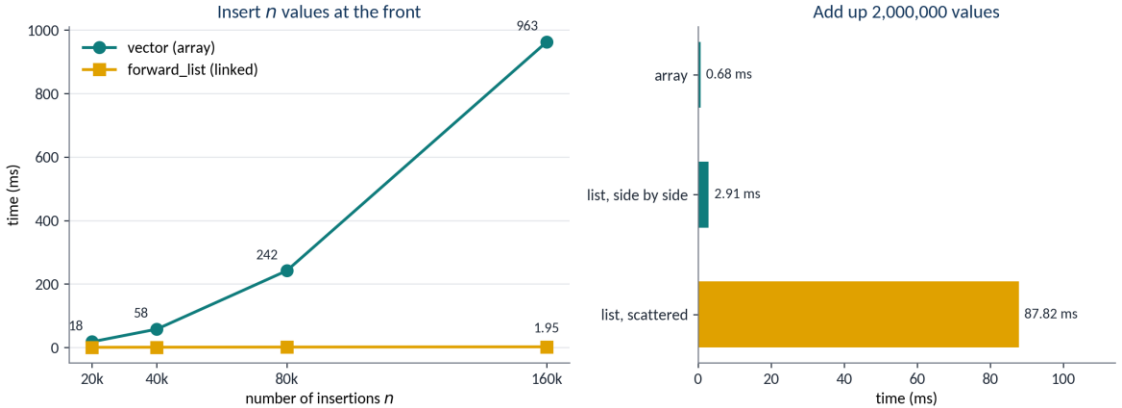


Figure 7.18 The measurements: front insertion is quadratic for the array and linear for the list (left); summing two million values is fastest in the array and about a hundred times slower in a list whose nodes are scattered (right).

Front insertion: doubling n multiplies the vector's time by about 4 (Equation (7.4)), the list's by about 2; at $n = 160{,}000$ the list is about 500 times faster.

Traversal: array 0.68 ms, side-by-side list 2.91 ms, scattered list 87.82 ms — all $\Theta(n)$; the cache decides.

***Figure 7.18**: for most workloads an array wins, because finding a position is a slow walk. Book: Section 7.12.4.*

Profitina in Practice

A feed that grows at the front: new posts go to the top — `frontInsertion`; keep the newest k with a `deque` of maximum length.

A queue of re-scans: due jobs wait in a FIFO queue — `enqueue` at the rear, `dequeue` at the front.

Taking turns fairly: round-robin over brands, the circular list.

A history that only grows: a weekly score trend — `backInsertion` with a tail and a forward traversal.

Identify which end the operations touch, and choose the structure whose cheap operations are exactly those. Book: Section 7.13.

Common Mistakes

Moving `head` to walk the list; the wrong loop condition (`aux`, `aux->next`, `aux->next->next`).

Dereferencing NULL; assignments in the wrong order (**Figure 7.17**); deleting before moving.

Forgetting to cut the last link, or the second pointer (`tail`, `rear_`).

A NULL test to stop a circular traversal; leaking the list; believing a correct output.

Recursing over a very long list; choosing a list for random access (Equation (7.3)).

Book: Section 7.14.

Summary: Five Things to Remember

1. A node holds data and `next`; the list is reached from `head` and ends at NULL. Access by position is $\Theta(n)$; change at a held position is $O(1)$.

2. Insert: link the new node to its successor, then its predecessor to it. Delete: save, move every pointer past, delete last.

3. A dummy node or a pointer to the link removes the first-node special cases; a tail gives $O(1)$ back insertion, never back deletion.

4. A circular list through `last` reaches both ends; linked stacks and queues are $O(1)$ and never full.

5. Prove memory safety with Valgrind or ASan; prefer arrays unless the operations are at positions already held.

Book: Section 7.15.

Next Time

Chapter 8 — Evaluation and Discussion II: Stacks, Pointers, and Linked Lists at Work. Twelve problems from Chapters 5, 6 and 7, solved from a blank page.

Draw the pointers first, trace by hand, read three programs, and study a tempting wrong solution under a memory checker.

Before then: work Exercises 7.1, 7.2, 7.5 and 7.9, and run one bug of `memory_bugs.cpp` under AddressSanitizer.

Bring the four shapes of the checklist: empty, one node, first, middle, last. Book: Section 7.16.

Lecture 8 — Evaluation and Discussion II: Stacks, Pointers, and Linked Lists at Work

Course: Data Structure (EF234201)

Book: Irfan Subakti, *Data Structure with C++, Java and Python*, Chapter 8

Lecturer: Irfan Subakti — yifana@gmail.com

No new structure today: twelve problems on Chapters 5, 6 and 7, each drawn before it is coded, traced by hand, solved in C++ (with Java and Python), broken by a tempting wrong solution, and closed with a check-yourself variant. Book: Chapter 8, Sections 8.1–8.15.

What This Meeting Covers

1. **A method** — seven steps for any problem in which boxes point at other boxes.
2. **Five connected problems** — completing an array stack, walking a two-way list, evens up and odds down, reversing words, removing duplicates.
3. **Seven new problems** — matching brackets, a circular queue, infix to postfix, three ways to pass an argument, in-place reversal, merging sorted lists, a linked queue.
4. **Looking back** — the problems side by side, Profitina, the common mistakes.

For the memory bugs, the failure is reported by a memory checker, and we read its report together. Book: Section 8.1.

Where We Stand

Table 8.1 The twelve problems of this chapter, the section that solves each one, and the chapter that supplies its main idea.

Problem	Section	What it asks	Main tool from
8.1	8.3	Complete an array-based `Stack` class	Ch. 5
8.2	8.4	Walk a two-way linked list by the directions stored in its nodes	Ch. 6, 7 (and a first look at Ch. 9)
8.3	8.5	Print the even numbers ascending, then the odd ones descending, using linked lists only	Ch. 6, 7
8.4	8.6	Reverse the words of a sentence with a linked list	Ch. 5, 7
8.5	8.7	Remove the nodes whose value has already appeared	Ch. 7
8.6	8.8	Check that the brackets of an expression match	Ch. 5
8.7	8.8	A circular queue of borrow requests	Ch. 5
8.8	8.8	Infix to postfix, then evaluate	Ch. 5
8.9	8.9	Pass by value, by reference and by pointer, traced	Ch. 6
8.10	8.10	Reverse a linked list in place	Ch. 6, 7
8.11	8.10	Merge two sorted linked lists	Ch. 7
8.12	8.10	A queue built from a linked list with `head` and `tail`	Ch. 5, 7

Table 8.1: Chapter 5 gave the stack and the queue, Chapter 6 the pointer and the three ways to pass an argument, Chapter 7 the singly linked list. Book: Section 8.1.

The One Idea Behind Almost Every Problem

> **THE ONE IDEA.** *Before you change a link, make sure you still have a way to reach everything you will need after the change.*

Stack: read the top item before you move `top` away from it.

Walk by directions: save the step count before the pointer leaves the node that holds it.

List reversal: save `cur->next` before you overwrite it.

Deletion: link around a node before you free it. **Queue:** reset `tail` when the last node disappears.

When a program today goes wrong, it is almost always because this rule was broken, and the wrong solutions break it in all of these ways. Book: Section 8.1.

How to Approach a Pointer Problem

Table 8.2 The seven-step method for a problem about stacks, queues and linked lists. Steps 1 and 2 understand the problem, Steps 3 to 6 design the pointer moves, and Step 7 checks them.

Step	Write on paper	Question it answers
1	The problem in your own words, and one example by hand	Do I know what the output must be?
2	The structure, and one sentence of reason	Stack, queue, array, one link or two?
3	Boxes, arrows, and every pointer variable	Who points where right now?
4	The invariant	What must stay true after every operation?
5	The edge cases	Empty, one node, first, last, equal neighbors?
6	The pointer assignments, numbered	Is anything lost if I do them in this order?
7	Tests, then the same tests under a memory checker	Does it work, and does it only *seem* to work?

Table 8.2: draw every node as a box with a data part and a pointer part, every pointer variable as a labeled arrow, and `nullptr` *explicitly. The edge cases are almost always the same: empty, one node, first, last, equal neighbors. Book: Section 8.2.*

One Node, Three Languages

Table 8.3 The same ideas in the three languages of this book. A Java reference and a Python name behave like a C++ pointer that can only be followed, never incremented, and whose target is freed automatically.

Idea	C++	Java	Python
A node type	`struct Node { int data; Node *next; };`	`class Node { int data; Node next; }`	`class Node:` with `self.data`, `self.next`
A variable that refers to a node	`Node *p` (a pointer)	`Node p` (a reference)	`p` (a name)
"Points nowhere"	`nullptr`	`null`	`None`
Create a node	`new Node{5, nullptr}`	`new Node(5, null)`	`Node(5)`
Follow the link	`p->next`	`p.next`	`p.next`
Destroy a node	`delete p;` (your job)	automatic (garbage collector)	automatic (reference counting and a collector)
Let a function change the caller's pointer	`Node *&head` (a reference to a pointer)	return the new value	return the new value
Library stack / queue	`std::stack`, `std::queue`	`ArrayDeque`	`list`, `collections.deque`

Table 8.3: only C++ makes you destroy nodes, so only C++ programs can leak or use a freed node. Java and Python cannot change the caller's variable, so they return the new first node: `head = insertSorted(head, ...)`*. Every C++ program was also run under AddressSanitizer and Valgrind: no warning, no invalid access, no leaked byte. Book: Section 8.2.*

Problem 8.1: Completing an Array Stack

PROBLEM 8.1. Complete the program: implement (a) `pop()`, (b) `push()`, (c) `peek()` and (d) `isEmpty()` for a class with `int a[MAX]` and `top`, whose constructor sets `top = -1`. The driver pushes 10, 20, 30 and pops once.

DEFINITION. The *stack invariant*: the items are exactly `a[0]`, ..., `a[top]`, bottom to top, and $-1 \le$ `top` $\le$ `MAX` -1.

$$\text{size} = \text{top} + 1, \text{empty} \Leftrightarrow \text{top} = -1, \text{full} \Leftrightarrow \text{top} = \text{MAX} - 1. \tag{8.1}$$

Equation (8.1): `top = -1` decides everything: `top` is the index of the top item. Check it at both ends: size 0 with no items, 1000 when full. Book: Sections 8.3.1 and 8.3.2.

Problem 8.1: By Hand

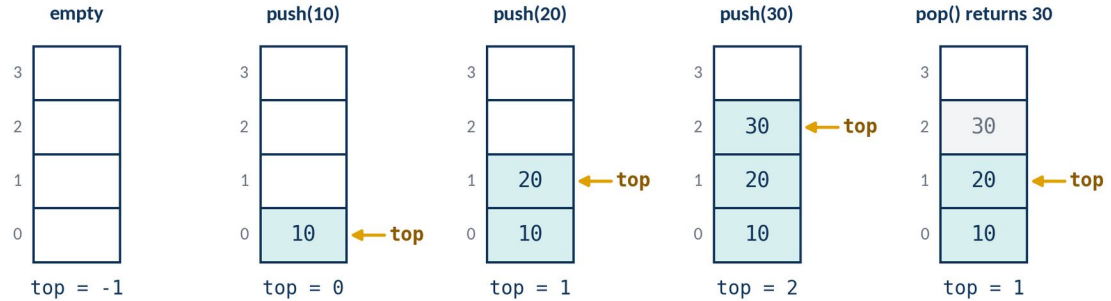


Figure 8.1 The array stack of Problem 8.1 after each operation of the driver: `top` is the index of the top item, and a pop only moves `top`; the value 30 stays in the array but is no longer part of the stack.

Figure 8.1: `push` is "increase, then store"; `pop` is "read, then decrease". After the pop, 30 is still in `a[2]` but no longer between `a[0]` and `a[top]`, so it is no longer an item. Book: Section 8.3.3.

Problem 8.1: The Solution

Listing 8.1 From `p801_stack.cpp`: `push` and `pop`, completed.

```
bool Stack::push(int x) {
    if (top >= MAX - 1) {          // a[MAX-1] is already in use: full
        cout << "Stack overflow: " << x << " not pushed\n";
        return false;
    }
    a[++top] = x;                  // FIRST move top up, THEN store
    return true;
}

int Stack::pop() {
    if (isEmpty()) {
        throw underflow_error("pop from an empty stack");
    }
    return a[top--];               // FIRST read a[top], THEN move top down
}
```

```
30 popped from stack
peek -> 20, size 2
pop -> 20
pop -> 10
```

Listing 8.1: an empty stack throws, since 0 or -1 are valid items. `peek` returns `a[top]`; all $O(1)$. Book: Sections 8.3.4 and 8.3.5.

Problem 8.1: The Tempting Wrong Solution

```
int pop() { return a[--top]; }   // WRONG: moves top first, then reads
```

```
20 popped from stack
10 popped from stack
isEmpty -> false
```

The first pop moves `top` from 2 to 1 *before* reading and returns 20, the item *under* the top; 30 is never returned.

The mirror mistake in `push`, `a[top++] = x`, writes to `a[-1]`: undefined behavior, caught by AddressSanitizer.

> **CHECK YOURSELF.** With `top = 0` ("next free cell"): `push` is `a[top++] = x`, `pop` is `return a[--top];`, `peek` is `a[top - 1]`, `isEmpty` is `top == 0`, and size = `top`. Both conventions work; mixing them never does.

Book: Sections 8.3.6 and 8.3.7.

Problem 8.2: Walking a Two-Way List by Directions

PROBLEM 8.2. Each input line is *direction step data*: `b` backward or `f` forward, the number of nodes to move, and the letter in the node; input stops at direction `0`. Print the letters met on the walk. Sample: `f 3 s`, `f 3 k`, `f 4 r`, `b 1 t`, `f 3 d`, `f 3 t`, `b 5 u`, `b 2 a`, `0 0 0` gives `strukdat`.

Two arrows per node: `next` and `prev` — a first look at Chapter 9.

Moving backward is `cur = cur->prev`. Appending sets **two** arrows: `last->next = newNode` *and* `newNode->prev = last`.

DEFINITION. The *two-way list invariant*: for every node `p` with a successor, `p->next->prev == p`; the first `prev` and the last `next` are `nullptr`.

Book: Sections 8.4.1 and 8.4.2.

Problem 8.2: The List Built From the Sample

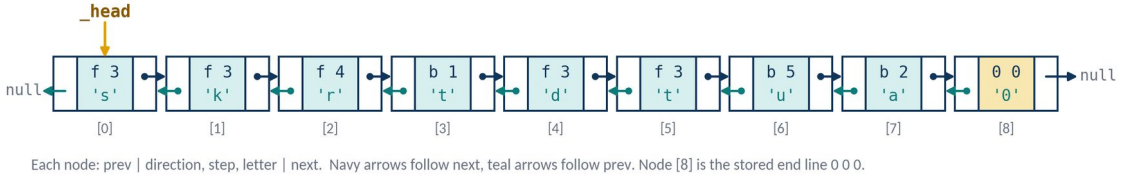


Figure 8.2 The two-way list built from the sample input of Problem 8.2: nine nodes, each with a `prev` and a `next` pointer. Node 8 holds the end line `0 0 0`, which the given reading loop also stores.

Figure 8.2: the given `do`–`while` reads a line, pushes it, and only then tests for `'0'`, so the end line is a node too: nine nodes, not eight. The walk stops when it arrives at that node. Book: Section 8.4.2.

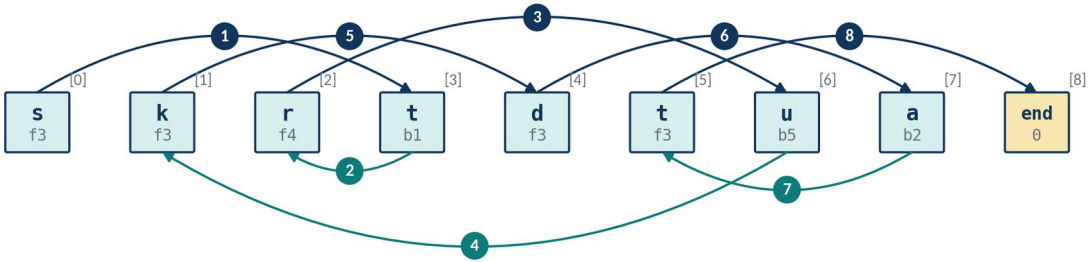
Problem 8.2: The Walk, by Hand

Table 8.4 The walk of Problem 8.2 on the sample input. Each row prints the current node's letter and then moves by the current node's own instruction.

Move	At node	Prints	Instruction	Lands on
1	0	`s`	`f 3`: forward 3	0 → 1 → 2 → 3
2	3	`t`	`b 1`: backward 1	2
3	2	`r`	`f 4`: forward 4	6
4	6	`u`	`b 5`: backward 5	1
5	1	`k`	`f 3`: forward 3	4
6	4	`d`	`f 3`: forward 3	7
7	7	`a`	`b 2`: backward 2	5
8	5	`t`	`f 3`: forward 3	8
—	8	—	direction `'0'`: stop	—

Table 8.4: print the current letter, then move by the current node's own instruction. The "Prints" column reads `strukdat`. Book: Section 8.4.3.

Problem 8.2: The Eight Moves



Moves 1 to 8 in order; printed letters s, t, r, u, k, d, a, t. Above: forward (next); below: backward (prev).

Figure 8.3 The eight moves of the walk: forward moves above the list, backward moves below, numbered in the order they happen. Every letter node is visited exactly once.

Figure 8.3: forward moves above, backward below; every letter node is visited exactly once — a property of this input, not of every input. Book: Section 8.4.3.

Problem 8.2: What Can Go Wrong in the Input

No end line: `cin` fails and the loop pushes forever; test whether reading succeeded.

A move runs off the end: check `cur != nullptr` inside the moving loop.

The directions go around in a circle: `f 1` next to `b 1`. Where the walk goes depends *only* on the current node, so a repeated node repeats forever.

$$\text{visits} > n \Rightarrow \text{some node was visited twice} \Rightarrow \text{the walk never reaches the end}. \tag{8.2}$$

Equation (8.2) is the pigeonhole principle: more visits than nodes means some node was visited twice. One counter and one comparison per visit turn a program that could hang into one that always stops. Book: Section 8.4.4.

Problem 8.2: The Solution

Listing 8.2 From `p802_deque_walk.cpp`: `traverse`, with the two guards.

```
    while (cur != nullptr && cur->direction != '0') {
        if (++visits > myList->_size) {   // a node came around twice: endless loop
            cout << " [the directions go around in a circle]";
            break;
        }
        cout << cur->data;
        int step = cur->step;             // SAVE the instructions of this node
        char direction = cur->direction;  // before cur moves away from it
        for (int k = 0; k < step && cur != nullptr; k++) {
            cur = (direction == 'b') ? cur->prev : cur->next;
        }
        if (cur == nullptr) {
            cout << " [the walk left the list]";
        }
    }
```

```
strukdat
list
aba [the directions go around in a circle]
xy [the walk left the list]
```

Listing 8.2: building costs $\Theta(n^2)$ without a tail; the walk costs 24 moves on the sample. Book: Section 8.4.5.

Problem 8.2: The Tempting Wrong Solution

```
  for (int k = 0; k < cur->step; k++)
      cur = (cur->direction == 'b') ? cur->prev : cur->next;   // WRONG
```

```
stttttttttttttttttt
```

`cur` changes inside the loop, so `cur->step` and `cur->direction` are read from a *different node* on every pass; the walk is trapped between nodes 2 and 3.

The repair: **copy what you need out of a node before you leave it** — `step` and `direction` into locals.

> **CHECK YOURSELF.** `f 2 l`, `f 2 s`, `b 1 i`, `f 1 t`, `0 0 0` prints `list`. Changing the second line to `f 4 s` sends the walk to a node that does not exist: `lis [the walk left the list]`.

Book: Sections 8.4.7 and 8.4.8.

Problem 8.3: Evens Up, Odds Down, Linked Lists Only

PROBLEM 8.3. Read N values; print the even ones ascending, then the odd ones descending, with linked lists and no arrays. Input 4, 32, 34, 543, 3456, 654, 567, 87, 6789, 98 gives 4, 32, 34, 98, 654, 3456, 6789, 567, 543, 87.

Insertion sort on a linked list, no shifting: `evens` ascending, `odds` descending, **one** insertion function receiving the order as a *pointer to a function*, `bool (*before)(int, int)`.

$$\sum_{k=0}^{n-1} k = \frac{n(n-1)}{2} = \Theta(n^2). \tag{8.3}$$

Equation (8.3): 45 steps at most for ten values; for a million, merge sort. Book: Sections 8.5.1–8.5.3.

Problem 8.3: Inserting 98

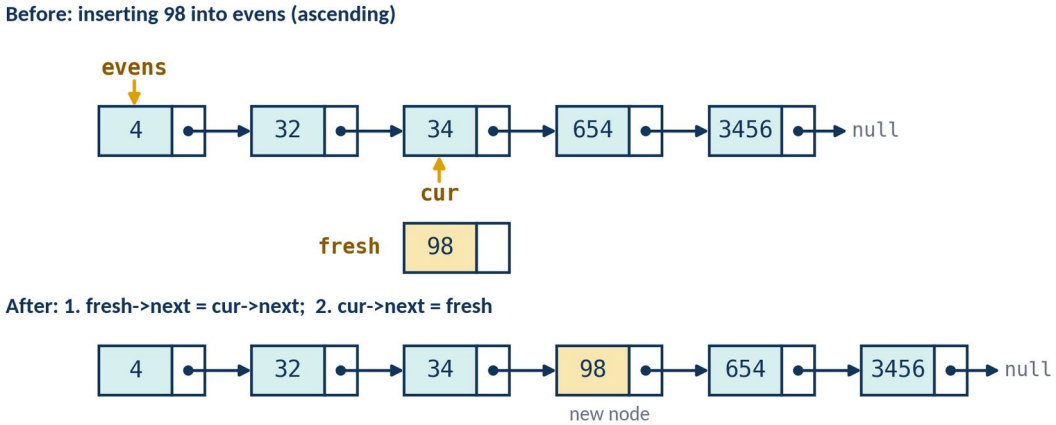


Figure 8.4 Inserting 98 into the ascending list `evens`: `cur` stops on the last node that stays in front (34); the new node is linked to 654 first, and only then is 34 linked to the new node.

`cur` looks at `cur->next`, so it stays on the **predecessor**, where the change happens.

Link the new node to 654 first, *then* 34 to the new node; the other order loses 654 and 3456.

`!before(value, cur->next->data)` puts equal values after their equals: **stable**.

Figure 8.4. Book: Section 8.5.3.

Problem 8.3: The Ten Insertions

Table 8.5 The ten insertions of the sample input of Problem 8.3. Printing `evens` and then `odds` gives the expected output.

Value	Even ?	Goes to	List after the insertion
4	yes	`evens`, empty list	4
32	yes	`evens`, after 4	4, 32
34	yes	`evens`, after 32	4, 32, 34
543	no	`odds`, empty list	543
3456	yes	`evens`, at the end	4, 32, 34, 3456
654	yes	`evens`, after 34	4, 32, 34, 654, 3456
567	no	`odds`, new first (567>543)	567, 543
87	no	`odds`, at the end	567, 543, 87
6789	no	`odds`, new first	6789, 567, 543, 87
98	yes	`evens`, after 34	4, 32, 34, 98, 654, 3456

Table 8.5: printing `evens`, then `odds`, gives the expected output. Book: Section 8.5.3.

Problem 8.3: The Solution

Listing 8.3 From `p803_evens_odds.cpp`: one sorted insertion for both orders.

```cpp
void insertSorted(Node *&head, int value, bool (*before)(int, int)) {
    Node *fresh = new Node{value, nullptr};
    if (head == nullptr || before(value, head->data)) {
        fresh->next = head;                 // the new node becomes the first
        head = fresh;
        return;
    }
    Node *cur = head;                       // find the last node that stays in front
    while (cur->next != nullptr && !before(value, cur->next->data)) {
        cur = cur->next;
    }
    fresh->next = cur->next;                // 1. the new node points to its successor
    cur->next = fresh;                      // 2. its predecessor points to it
```

```
-2
0
4
4
9
```

Listing 8.3: `Node *&head` *lets* `head = fresh` *reach* `main`*. Java passes a lambda; Python the function. Zero is even. Book: Sections 8.5.4 and 8.5.5.*

Problem 8.3: The Tempting Wrong Solution

```
if (x % 2 == 1) insertSorted(odds, x, descending);    // WRONG for negative x
else            insertSorted(evens, x, ascending);
```

```
-7
-3
-2
0
4
4
9
```

Since C++11, `/` truncates toward zero, so `-7 % 2` is -1, not 1: -7 and -3 join the evens. Java agrees; Python's `%` gives 1. Test parity with `x % 2 == 0` and let "odd" be everything else.

CHECK YOURSELF. For 5, 0, -4, 15, -9, 8: correct -4, 0, 8, 15, 5, -9; wrong -9, -4, 0, 8, 15, 5. Six nodes created, six freed.

Book: Sections 8.5.6 and 8.5.7.

Problem 8.4: Reversing the Words of a Sentence

PROBLEM 8.4. Reverse a string using a linked list. Input `practice makes a man perfect`; output `perfect man a makes practice`.

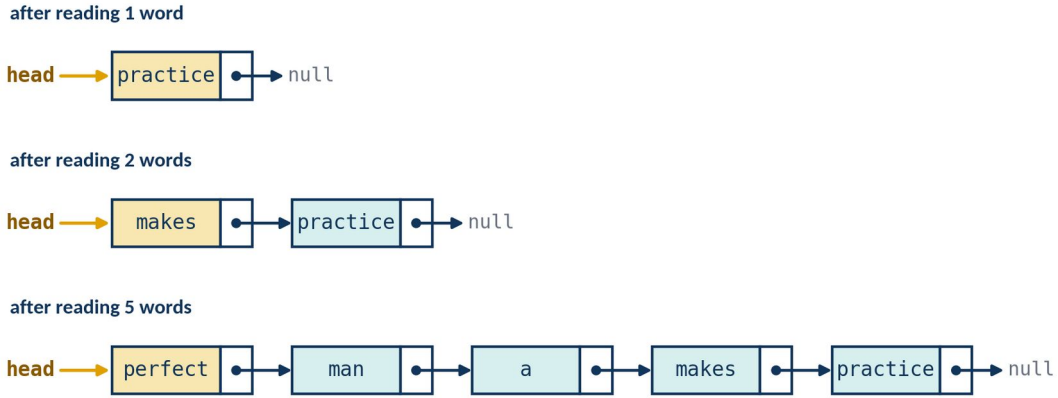


Figure 8.5 Building the list of Problem 8.4 by front insertion: each new word becomes the first node, so the list reads the sentence backward.

Figure 8.5: "last word read, first word printed" is a stack, and `frontInsertion` *is a push.* `head = new WordNode{w, head};` *evaluates the right side first, so the new node receives the old* `head`*. The reversal happens during the building. Book: Sections 8.6.1–8.6.3.*

Problem 8.4: The Solution

Listing 8.4 From `p804_reverse_words.cpp`: `getline`, `istringstream`, push, then print and free.

```
    while (words >> w) {                    // one word at a time, spaces skipped
        pushFront(head, w);                 // the last word read ends up first
    }

    bool first = true;
    while (head != nullptr) {
        if (!first) cout << ' ';            // a space BETWEEN words, never at the end
        cout << head->word;
        first = false;
        WordNode *gone = head;
        head = head->next;
        delete gone;
    }
```

```
$ ./p804 < p804.txt
perfect man a makes practice
$ ./p804 < p804_check.txt
friends are queues and stacks
```

Listing 8.4: `>>` *skips any number of spaces; the flag* `first` *puts a space between words, never at the end.* $\Theta(L)$ *for a line of* L *characters. Book: Sections 8.6.4 and 8.6.5.*

Problem 8.4: The Tempting Wrong Solution

```
char str[100];                              // WRONG: assumes a short sentence
int i = 0, c;
while ((c = getchar()) != '\n' && c != EOF)
    str[i++] = (char)c;                     // no check against the size of str
str[i] = '\0';
```

```
wrong/p804_wrong.cpp:15:16: runtime error: index 100 out of bounds for type 'char [100]'
...
==...==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x...
WRITE of size 1 at 0x... thread T0
    #0 0x... in main wrong/p804_wrong.cpp:15
```

A sentence of 100 characters or more writes past `str`; nothing obliges the program to crash. AddressSanitizer stops at the first bad write, line 15. The repair: `std::string` with `getline`.

> **CHECK YOURSELF.** One word, `queue`: one node, prints `queue`. An empty line: no nodes. With `backInsertion` in place of `pushFront`: `practice makes a man perfect`, unchanged.

Book: Sections 8.6.6 and 8.6.7.

Problem 8.5: Removing Duplicate Values From a List

> **PROBLEM 8.5.** Complete `remove_duplicate()`, which removes the nodes with the same data. Input 1, 2, 3, 4, 1, 5, 6, 3, 2, −1; output `1 2 3 4 1 5 6 3 2 -1`, then `1 2 3 4 5 6`.

Keep the **first** appearance of every value. The stored end marker -1 **is not data**: a small `slist_popBack` drops it.

$$\sum_{k=0}^{n-1} k = \frac{n(n-1)}{2} = \Theta(n^2) \text{ comparisons, and } O(1) \text{ extra memory}. \tag{8.4}$$

Equation (8.4): a set of seen values would give $O(n)$ time for $O(n)$ memory (Chapter 15). Book: Sections 8.7.1–8.7.3.

Problem 8.5: Unlink, Then Free, Then Do Not Move

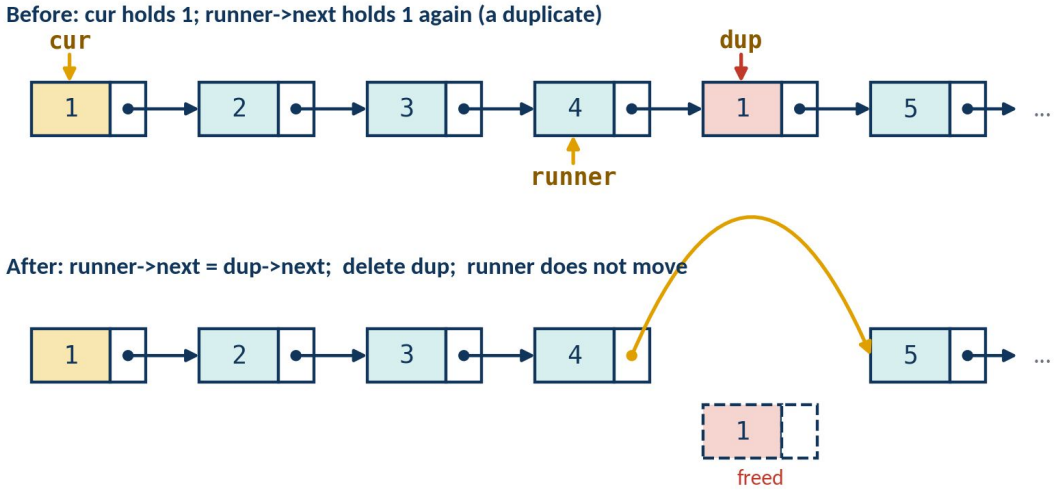


Figure 8.6 Deleting a later copy: `runner` stands on the predecessor of the duplicate, links around it, and only then frees it; `runner` does not move, because its new successor has not been examined yet.

1. `dup = runner->next;` **2.** `runner->next = dup->next;` **3.** `delete dup;`

After a deletion `runner` **stays**, because its new `next` has not been examined.

Figure 8.6: freeing before step 2 would read `dup->next` from freed memory. Book: Section 8.7.3.

Problem 8.5: The Solution

Listing 8.5 From `p805_remove_duplicates.cpp`: `remove_duplicate` with `cur` and `runner`.

```
void remove_duplicate(linkedList *myList) {
    for (SListNode *cur = myList->_head; cur != nullptr; cur = cur->next) {
        SListNode *runner = cur;           // runner->next is the node being examined
        while (runner->next != nullptr) {
            if (runner->next->data == cur->data) {
                SListNode *dup = runner->next;
                runner->next = dup->next;  // 1. unlink: bridge over the duplicate
                delete dup;                // 2. only then free it
                myList->_size--;           // runner stays: its NEW next is unexamined
            } else {
                runner = runner->next;     // nothing removed: move on
            }
        }
    }
}
```

```
$ ./p805 < p805.txt
1 2 3 4 1 5 6 3 2 -1
1 2 3 4 5 6
nodes left: 6
```

Listing 8.5: the `else` branch is the only place where `runner` moves. Java and Python need only step 2. Book: Sections 8.7.3–8.7.5.

Problem 8.5: The Tempting Wrong Solution

```
  if (runner->next->data == cur->data) {
      SListNode *dup = runner->next;
      runner->next = dup->next;
      delete dup;
  }
  runner = runner->next;          // WRONG: also moves after a removal
```

```
7 7 7 2 7 2 -1
7 7 2
nodes left: 3
```

On `7 7 7 2 7 2 -1` the third 7 escapes: after a deletion the wrong line moves `runner` onto a node it never compared. The sample has no neighboring copies, so it passes.

> **CHECK YOURSELF.** On the ten-node sample `remove_duplicate` makes $9+7+5+3+2+1+0=27$ comparisons, below the 45 of Equation (8.4), because each deletion shortens every later scan.

Book: Sections 8.7.6 and 8.7.7.

Problem 8.6: Matching Brackets

> **PROBLEM 8.6.** Decide whether the brackets `()`, `[]` and `{}` of a line are balanced; if not, say what is wrong and where. Test `(a+b)*[c-d]`, `{[()()]}`, `([)]`, `((x)`, `x)]` and the empty line.

Every closer must match the **most recent** opener still open: last in, first out — a stack.

Three ways to fail: a closer on an empty stack, a closer that does not match the top, openers left at the end.

For `([)]`: `(` and `[` are pushed; `)` meets `[` on top.

The stack's capacity is chosen at run time (`new char[capacity]`); its destructor calls `delete[]` on every return path, and copying is forbidden. $\Theta(L)$ time. Book: Section 8.8.1.

Problem 8.6: The Solution

Listing 8.6 From `p806_brackets.cpp`: `check`, with its three failure messages.

```
        if (c == '(' || c == '[' || c == '{') {
            st.push(c, pos);
        } else if (c == ')' || c == ']' || c == '}') {
            if (st.isEmpty()) {
                return string("'") + c + "' at " + to_string(pos) + " has no opening partner";
            }
            if (st.symbol[st.top] != partnerOf(c)) {
                return string("'") + c + "' at " + to_string(pos) + " cannot close '" +
                       st.symbol[st.top] + "' from " + to_string(st.where[st.top]);
            }
            st.pop();
        }
    }
```

```
"([)]" -> ')' at 3 cannot close '[' from 2
"((x)" -> '(' at 1 is never closed
"x)]" -> ')' at 2 has no opening partner
```

Listing 8.6: positions count from 1, so the messages point at the exact character. Book: Section 8.8.1.

Problem 8.6: The Tempting Wrong Solution

Counting: add one for an opener, subtract one for a closer, fail below zero, succeed at zero.

```
"(a+b)*[c-d]" -> balanced
"{[()()]}" -> balanced
"([)]" -> balanced
"((x)" -> not balanced
"x)]" -> not balanced
"(]" -> balanced
```

A counter knows *how many* brackets are open, not *which ones*, so it accepts `([)]` and even `(]`. Remembering an order is what a stack does.

> **CHECK YOURSELF.** `{(a)[b]}`: the stack is empty at position 8, so `'}' at 9 has no opening partner`. `[(])`: `']' at 3 cannot close '(' from 2`.

Book: Section 8.8.1.

Problem 8.7: A Circular Queue of Borrow Requests

PROBLEM 8.7. A queue of four borrow requests in an array, reusing slots freed by `dequeue`: enqueue `B1001` to `B1005`; dequeue twice; enqueue `B1005` and `B1006`; dequeue everything.

Store `front` (the oldest request) and `count` (how many wait); everything else follows.

$$\text{rear} = (\text{front} + \text{count}) \bmod \text{CAPACITY}\,, \quad \text{front} \leftarrow (\text{front} + 1) \bmod \text{CAPACITY} \text{ after a dequeue}\,. \tag{8.5}$$

Empty is `count == 0`, full is `count == CAPACITY`: they can never be confused.

Equation (8.5): the remainder bends the array into a ring. With only `front` and `rear`, a full ring and an empty ring look the same. Book: Section 8.8.2.

Problem 8.7: The Ring

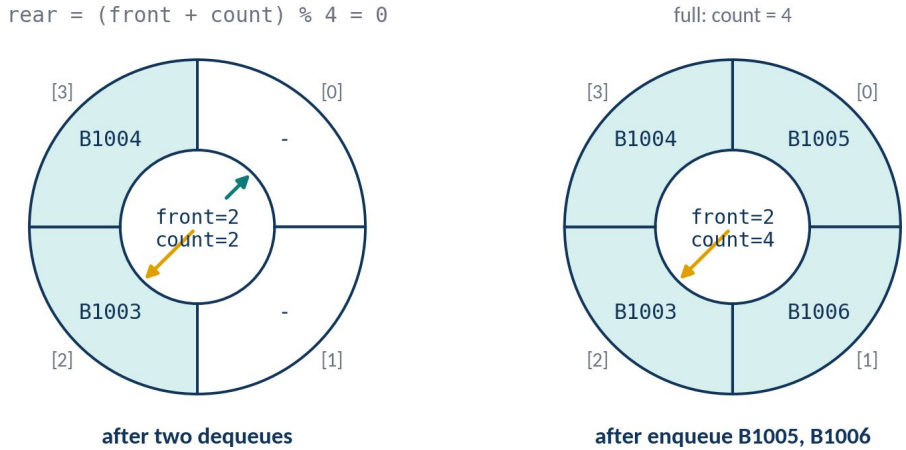


Figure 8.7 The circular queue of Problem 8.7 drawn as a ring. The amber arrow is `front`, the teal arrow is the next free slot `rear`. After two dequeues (left) the next request goes into cell 0; after two more enqueues (right) the ring is full, and the queue order starts at cell 2 and wraps around.

Figure 8.7: after two dequeues, $(2+2) \bmod 4 = 0$, so `B1005` goes into the cell `B1001` left; the array reads `B1005 B1006 B1003 B1004`, but the queue order starts at `front` = 2 and wraps around. Book: Section 8.8.2.

Problem 8.7: The Solution

Listing 8.7 From `p807_circular_queue.cpp`: `enqueue` and `dequeue` with `front` and `count`.

```
    bool enqueue(const string &id) {
        if (isFull()) return false;
        int rear = (front + count) % CAPACITY;   // the first free slot, wrapping around
        slot[rear] = id;
        count++;
        return true;
    }

    string dequeue() {                           // precondition: !isEmpty()
        string id = slot[front];
        slot[front] = "-";                       // only to make the picture clearer
        front = (front + 1) % CAPACITY;          // the next oldest, wrapping around
        count--;
        return id;
    }
```

```
  slots [B1005 B1006 B1003 B1004]  front=2 count=4
served in order: B1003 B1004 B1005 B1006
  slots [- - - -]  front=2 count=0
```

Listing 8.7: every operation is $O(1)$; shifting after each dequeue would cost $O(n)$. At the end `count` is 0 and `front` has stopped at 2, which does not matter. Book: Section 8.8.2.

Problem 8.7: The Tempting Wrong Solution

The linear queue of Chapter 5 is full when `rear` reaches the last cell:

```
dequeue -> B1001
dequeue -> B1002
waiting: 2 of 4
enqueue B1005 -> refused
```

Two cells are free and the request is refused: “false full”. The test asks where the rear *is*; the question is how many items are *waiting*.

> **CHECK YOURSELF.** Capacity 4: enqueue A, B, C; dequeue twice; enqueue D, E, F; dequeue. The slots end as `E F - D`, `front` 3, `count` 3; the order is D, E, F.

Book: Section 8.8.2.

Problem 8.8: From Infix to Postfix

PROBLEM 8.8. `toPostfix` converts an infix expression (single-letter or single-digit operands, `+ - * /`, parentheses) to postfix; `evaluate` computes a postfix expression of digits. Test `3+2*5`, `a+b*c-d`, `8-3-2`, `(1+2)*(7-3)/2`, `9-4/2*3`.

1. Operand → output. **2.** `(` → push. **3.** `)` → pop to the output down to `(`, then drop it.

4. Operator → first pop every operator of degree **greater than or equal to** its own, then push. **5.** At the end, pop everything.

Degree: `* /` 2, `+ -` 1, `(` 0. This is Dijkstra's *shunting-yard algorithm*.

"Greater than or equal": `8-3-2` is $(8-3)-2=3$*; with "greater than" only, the output would be `832--`, worth 7. Book: Section 8.8.3.*

Problem 8.8: `a+b*c-d` by Hand

Table 8.6 Converting `a+b*c-d` to postfix with the rules of Problem 8.8. The stack holds operators that are waiting for their right operand to be complete.

Read	Rule	Stack (top on the right)	Output so far
`a`	1		`a`
`+`	4: stack empty, push	`+`	`a`
`b`	1	`+`	`ab`
`*`	4: degree of `+` (1) < 2, push	`+ *`	`ab`
`c`	1	`+ *`	`abc`
`-`	4: pop `*` ($2 \geq 1$), pop `+` ($1 \geq 1$), push `-`	`-`	`abc*+`
`d`	1	`-`	`abc*+d`
end	5: pop everything		`abc*+d-`

*Table 8.6: the stack holds operators waiting for their right operand. For evaluation, a digit is pushed; an operator pops two numbers — the **first** popped is the **right** operand — and pushes the result. Book: Section 8.8.3.*

Problem 8.8: The Solution

Listing 8.8 From `p808_postfix.cpp`: `evaluate` with `std::stack`.

```
    for (char c : postfix) {
        if (isdigit(static_cast<unsigned char>(c))) {
            values.push(c - '0');
        } else {
            if (values.size() < 2) throw invalid_argument("missing operand");
            long long y = values.top();     // the FIRST pop is the RIGHT operand
            values.pop();
            long long x = values.top();     // the second pop is the left operand
            values.pop();
            if (c == '+') values.push(x + y);
            else if (c == '-') values.push(x - y);
            else if (c == '*') values.push(x * y);
            else {
                if (y == 0) throw invalid_argument("division by zero");
                values.push(x / y);
```

```
3+2*5  ->  325*+  =  13
8-3-2  ->  83-2-  =  3
evaluate("3+"): missing operand
```

Listing 8.8: a malformed postfix string throws with a message. Java uses `ArrayDeque`, Python a list; Python's `//` rounds down, so it restores the sign by hand. Both functions are $\Theta(L)$. Book: Section 8.8.3.

Problem 8.8: The Tempting Wrong Solution

Call the first number popped `x`, the second `y`, and compute `x - y`:

```
325*+  =  13
83-2-  =  7
12+73-*2/  =  0
942/3*-  =  -9
```

`325*+` still gives 13: addition and multiplication do not care. Subtraction and division do.

CHECK YOURSELF. `(a+b)*(c-d)/e` is `ab+cd-*e/`. `72-3*` is 15; `92/` is 4, truncated toward zero.

A test with `3+2*5` alone would pass the wrong program. Book: Section 8.8.3.

Problem 8.9: By Value, by Reference, by Pointer

PROBLEM 8.9. A member owes a fine of 1000 in `int balance`. Write three functions that add 500, by value, by reference and by pointer; print `balance` after each and whether the parameter is the same box. Add two `swap`s and a `larger` that returns an address.

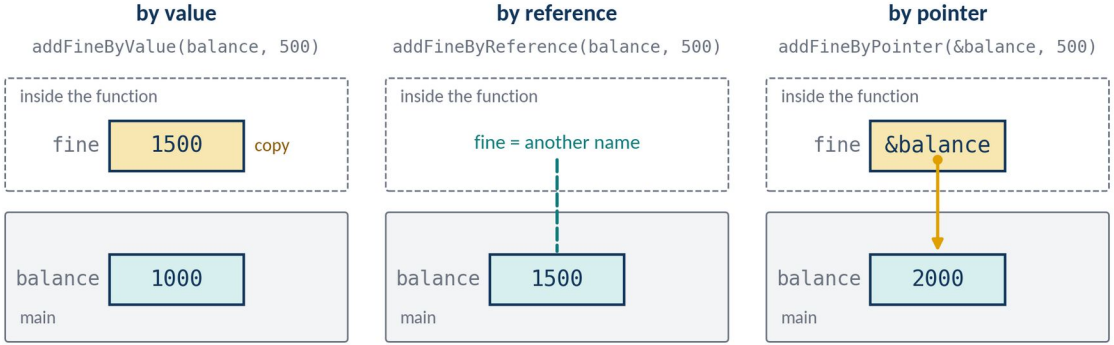


Figure 8.8 The three ways to pass `balance`, drawn at the end of each call. By value, the function works on a copy that disappears when it returns. By reference, `fine` is another name for `balance` itself. By pointer, `fine` holds the address of `balance`, and `*fine` follows it.

Figure 8.8: by value, a copy dies at return; by reference, `fine` is `balance` (and nothing at the call site warns you); by pointer, `&balance` is the visible warning, and `*fine` follows it. Book: Section 8.9.1.

Problem 8.9: The C++ Solution

Listing 8.9 From `p809_three_ways.cpp`: by value and by reference, each checking its own box.

```
void addFineByValue(int fine, int amount) {       // fine is a NEW box holding a copy
    cout << "  inside by value    : same box as balance? " << (&fine == callerBox) << "\n";
    fine += amount;                                // changes the copy only
}

void addFineByReference(int &fine, int amount) {  // fine is another NAME for balance
    cout << "  inside by reference: same box as balance? " << (&fine == callerBox) << "\n";
    fine += amount;
}
```

```
start: balance = 1000
  inside by value    : same box as balance? false
after by value    : balance = 1000
  inside by reference: same box as balance? true
after by reference: balance = 1500
  inside by pointer  : same box as balance? true
after by pointer  : balance = 2000
```

Listing 8.9: by pointer, `*fine += amount` gives 2000. `*larger(&x, &y) += 10`: the call is an address, and `*` opens it. Book: Section 8.9.1.

Problem 8.9: Java and Python

```
start: balance = 1000
after add_fine_int(balance, 500)    : balance = 1000
after add_fine_account(acc, 500)    : acc.balance = 1500
after replace(acc, 9999)            : acc.balance = 1500
same object? True
```

Table 8.7 What a function can change in each language. This is why the Java and Python versions of Problems 8.3 to 8.5 return the new first node, where the C++ versions take `Node *&head`.

Question	C++	Java	Python
Can a function change a caller's `int` variable?	yes, by reference or by pointer	no	no
Can it change an object the caller refers to?	yes	yes, through the reference	yes, if the object can change
Can it make the caller's variable refer to another object?	yes, with `T *&` or `T **`	no	no
How to hand back a new first node	reference to a pointer, or return it	return it	return it

Table 8.7: Java passes everything by value, but an object variable's value is a reference; Python parameters are new names for the same objects, and `int`s never change. So neither can change `balance`, both can change an `Account`, and `swap(int, int)` is impossible. Book: Section 8.9.1.

Problem 8.9: The Tempting Wrong Solution

```
int *makeFine(int amount) {
    int fine = amount;      // a local box: it disappears when the function returns
    return &fine;           // WRONG: the caller receives the address of a dead box
}
```

```
wrong/p809_wrong.cpp:7:12: warning: address of local variable 'fine' returned [-Wreturn-local-addr]
```

The caller holds a *dangling pointer*; with optimization GCC 13 even returns null. Return the value, or use `new` and say who deletes it.

> **CHECK YOURSELF.** `int a = 5, b = 7; int *p = &a; int &r = b; *p = *p + r; r = 1; p = &b; *p += a;` ends with `a` = 12 and `b` = 13.

Warnings like this one are why every program is compiled with `-Wall -Wextra`. Book: Section 8.9.1.

Problem 8.10: Reversing a List in Place

PROBLEM 8.10. Write `reverse`, which reverses a singly linked list *in place* — no node created or deleted — and returns the new first node. Trace it on 1, 2, 3, 4, 5; test one node and the empty list.

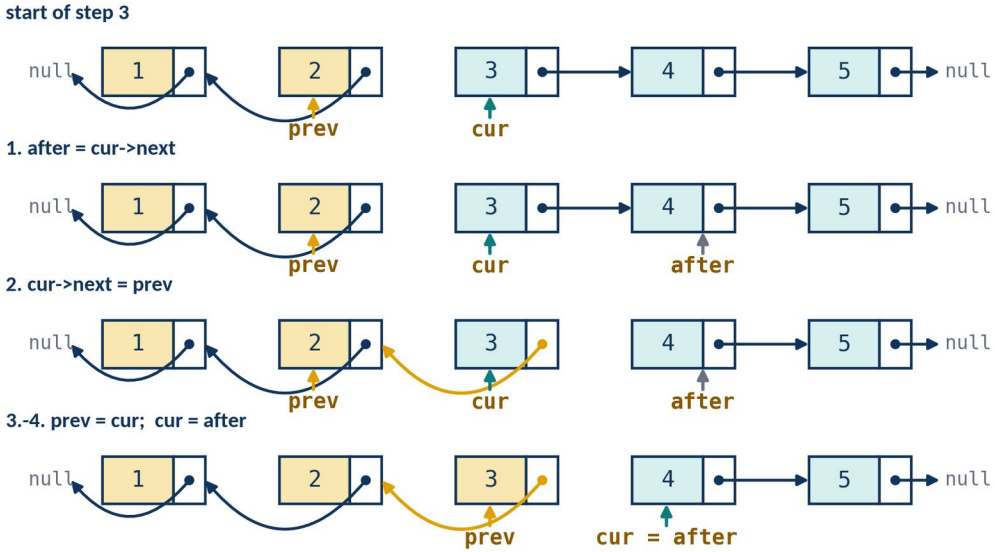


Figure 8.9 One step of the in-place reversal, the third one: save `cur->next` in `after`, turn `cur`'s arrow around to `prev`, then move `prev` and `cur` one node on. Amber nodes are the reversed part.

Figure 8.9. Book: Section 8.10.1.

Problem 8.10: The Solution

Listing 8.10 From `p810_reverse_list.cpp`: in-place reversal with three pointers (tracing lines left out).

```
Node *reverse(Node *head, bool trace) {
    Node *prev = nullptr;    // the part already reversed
    Node *cur = head;        // the part not yet touched
    int step = 0;
    while (cur != nullptr) {
        Node *after = cur->next;   // 1. SAVE the way on
        cur->next = prev;          // 2. turn cur's arrow around
        prev = cur;                // 3. the reversed part grows by one
        cur = after;               // 4. the rest shrinks by one
    }
    return prev;
}
```

```
start       : done = []   rest = [1 2 3 4 5]
after step 1: done = [1]   rest = [2 3 4 5]
after step 2: done = [2 1]   rest = [3 4 5]
after step 3: done = [3 2 1]   rest = [4 5]
after step 4: done = [4 3 2 1]   rest = [5]
after step 5: done = [5 4 3 2 1]   rest = []
result: [5 4 3 2 1]
```

Listing 8.10: one node and the empty list need no special code. Book: Section 8.10.1.

Problem 8.10: The Tempting Wrong Solution

```
  cur->next = prev;      // WRONG: the only way to node 2 is overwritten here ...
  prev = cur;
  cur = cur->next;       // ... so this now goes BACK to prev (nullptr at first)
```

```
1
==...==ERROR: LeakSanitizer: detected memory leaks
Direct leak of 16 byte(s) in 1 object(s) allocated from:
Indirect leak of 48 byte(s) in 3 object(s) allocated from:
SUMMARY: AddressSanitizer: 64 byte(s) leaked in 4 allocation(s).
```

After `cur->next = prev`, nothing points at node 2; `cur = cur->next` goes *back* to `nullptr`. The loop ends after one step, and LeakSanitizer reports node 2 lost directly and nodes 3–5 indirectly.

> **CHECK YOURSELF.** For 8 → 9: `reverse` returns 9, whose `next` is 8, whose `next` is `nullptr`. The loop runs twice, so each assignment runs twice.

Book: Section 8.10.1.

Problem 8.11: Merging Two Sorted Lists

PROBLEM 8.11. Branch A has 1003, 1007, 1010, 1015 and branch B has 1001, 1007, 1012, both ascending. Write `merge(a, b)` by relinking nodes, count the comparisons, and test empty and non-overlapping lists.

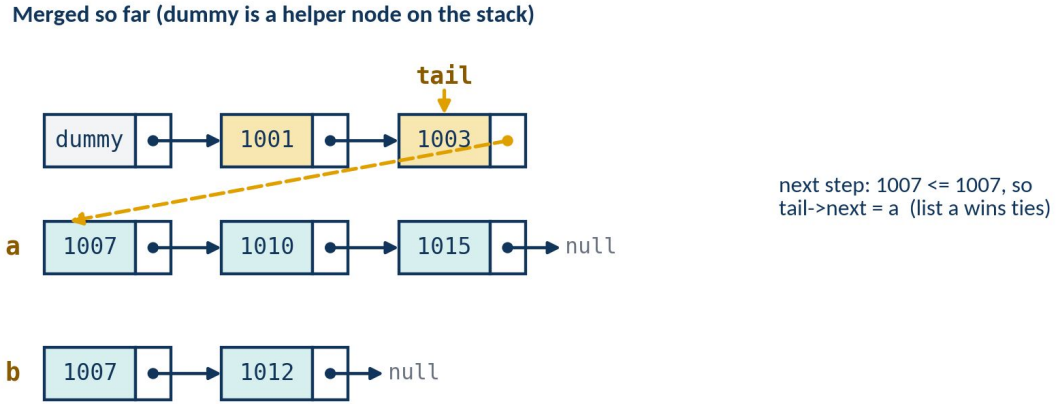


Figure 8.10 Merging the two branch lists: the result hangs from a dummy node, `tail` marks its last node, and the next comparison is between the front nodes of `a` and `b`.

*Figure 8.10: the smallest remaining identifier is always at one of the two fronts. A **dummy node** on the stack removes the "first node" case: `tail` starts on it, every attachment is `tail->next = node; tail = node;`, and the answer is `dummy.next`. Book: Section 8.10.2.*

Problem 8.11: The Solution

Listing 8.11 From `p811_merge_sorted.cpp`: the merge with a dummy node.

```
Node *merge(Node *a, Node *b) {
    Node dummy{0, nullptr};      // a helper node on the stack, never printed
    Node *tail = &dummy;         // tail = the last node of the merged list so far
    while (a != nullptr && b != nullptr) {
        comparisons++;
        if (a->id <= b->id) {    // <= : on a tie, list a goes first (stable)
            tail->next = a;
            a = a->next;
        } else {
            tail->next = b;
            b = b->next;
        }
        tail = tail->next;
    }
    tail->next = (a != nullptr) ? a : b;   // attach whatever is left, all at once
    return dummy.next;
}
```

```
A + B       : 1001 1003 1007 1007 1010 1012 1015  (6 comparisons)
A + empty   : 1003 1007 1010 1015  (0 comparisons)
empty+empty :  (0 comparisons)
low + high  : 1 2 3 7 8 9  (3 comparisons)
```

Listing 8.11: 6 comparisons for the branches, 0 with an empty list, 3 for `low + high`. `<=` makes the merge stable. Book: Section 8.10.2.

Problem 8.11: The Bound, and the Wrong Solution

$$\text{comparisons} \le m+n-1, \text{time} = \Theta(m+n), \text{extra memory} = O(1). \tag{8.6}$$

```
A + B       : 1001 1003 1007 1007 1010 1012  (6 comparisons)
lost        : 1015  (6 comparisons)
```

Wrong: forget line 27. The loop stops when *either* list is empty; the other still has nodes.

> **CHECK YOURSELF.** a = 2, 4, 6, 8 and b = 5: 2 vs 5, 4 vs 5, 6 vs 5; b is empty after 3 comparisons and 6, 8 are attached at once. The bound is $4+1-1=4$.

Equation (8.6): every comparison places one node. Book: Section 8.10.2.

Problem 8.12: A Queue Made of Linked Nodes

PROBLEM 8.12. Implement a queue of names with `head` (served) and `tail` (joining), both operations $O(1)$. Enqueue Ani, Budi, Citra; serve all three; enqueue Dewi and Eko; serve one. A `dequeue` on an empty queue is reported, not a crash.

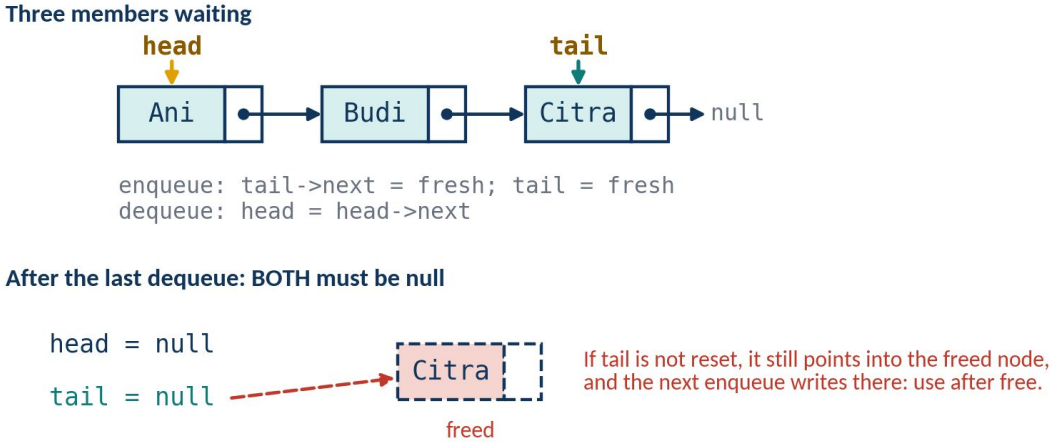


Figure 8.11 The linked queue of Problem 8.12. Enqueue works at `tail` and dequeue at `head`; when the last node leaves, `tail` must be reset to `null` too, or it is left pointing into freed memory.

*Figure 8.11: the invariant — `head` at the oldest node, `tail` at the newest, both `nullptr` when empty, `count` nodes. When the last node leaves, **`tail` must become `nullptr` too**. Book: Section 8.10.3.*

Problem 8.12: The Solution

Listing 8.12 From `p812_linked_queue.cpp`: `dequeue`, with the line that is easy to forget.

```
    string dequeue() {                                      // O(1)
        if (isEmpty()) throw underflow_error("dequeue from an empty queue");
        Node *gone = head;
        string name = gone->name;
        head = head->next;
        if (head == nullptr) {
            tail = nullptr;                                 // the queue is empty again
        }
        delete gone;
        count--;
        return name;
    }
```

```
  queue:  (size 0, head null, tail null)
  queue: Dewi Eko  (size 2, head Dewi, tail Eko)
served Dewi
caught: dequeue from an empty queue
```

Listing 8.12: `enqueue` *sets both pointers on an empty queue, else* `tail->next = fresh; tail = fresh;`*. The destructor dequeues what is left; copying is forbidden. Book: Section 8.10.3.*

Problem 8.12: The Tempting Wrong Solution

Wrong: drop the lines that reset `tail`, and test emptiness with `tail`. After the queue empties, `tail->next = fresh` writes into Citra's freed node.

```
==...==ERROR: AddressSanitizer: heap-use-after-free on address 0x...
WRITE of size 8 at 0x... thread T0
    #0 0x... in LinkedQueue::enqueue(...) wrong/p812_wrong.cpp:34
0x... is located 32 bytes inside of 40-byte region [0x...,0x...)
```

> **CHECK YOURSELF.** Enqueue X; dequeue; enqueue Y; enqueue Z; dequeue: `head` and `tail` both end on Z, `count` 1; after the first dequeue both are `nullptr`.

An 8-byte write 32 bytes into a 40-byte block: exactly the `next` field after a 32-byte `std::string` — the stale `tail->next`. Book: Section 8.10.3.

The Problems Compared, and Three Cost Patterns

A fixed number of pointer or index changes is $O(1)$: every stack and queue operation, array or nodes.

One walk is $\Theta(n)$: reversal, merging, printing, bracket matching, conversion to postfix.

A walk inside a walk is $\Theta(n^2)$ by the sum of Equation (8.3) and Equation (8.4): sorted insertion, duplicate removal, appending without a tail.

When the third group is too slow, ask which structure turns the inner walk into one step: a tail pointer (Chapter 9), a hash set (Chapter 15), a heap (Chapter 14).

Table 8.8 The twelve problems of the chapter: the structure each uses, its cost, and the input on which its tempting wrong solution fails. n is a number of nodes, L a length in characters.

Problem	Structure	Time	Extra memory	The input the wrong solution gets wrong
8.1	array stack	$O(1)$ per operation	$\Theta(\text{MAX})$	any `pop` (reads the item below the top)
8.2	two-way list	$\Theta(n^2)$ to build, one step per move	$O(1)$	any instruction whose next node has a different step
8.3	two sorted lists	$\Theta(n^2)$ worst case	one node per value	negative odd numbers
8.4	list as a stack	$\Theta(L)$	one node per word	a sentence of 100 characters or more
8.5	list, two pointers	$\Theta(n^2)$	$O(1)$	two copies of a value next to each other
8.6	array stack	$\Theta(L)$	$O(L)$	crossed pairs such as `( [ ) ]`
8.7	circular array	$O(1)$ per operation	$\Theta(\text{CAPACITY})$	enqueue after some dequeues
8.8	library stacks	$\Theta(L)$	$O(L)$	subtraction or division
8.9	function parameters	$O(1)$	$O(1)$	any call that should change the caller's variable
8.10	list, three pointers	$\Theta(n)$	$O(1)$	any list of two or more nodes
8.11	two sorted lists	$\Theta(m+n)$	$O(1)$	lists with different largest values
8.12	list with head and tail	$O(1)$ per operation	one node per item	enqueue after the queue has emptied

Table 8.8: read down the last column — the input that breaks the wrong solution is rarely the example in the statement. Book: Section 8.11.

Profitina in Practice

Work that waits its turn is a queue: scheduled re-scans and a real-time feed per viewer, first in, first out; a fixed-capacity circular queue bounds what one slow reader can consume.

Following references is walking a list: a deep reply back to its thread, like `prev` in Problem 8.2 — with deleted posts (`nullptr`) and loops (Equation (8.2)) to guard against.

Keep what you need before you change anything: the composer keeps the text before clearing the box, and restores it if sending fails.

The server is Python, so nobody frees a node by hand; the cousins of these bugs — a stale "last item" reference — are caught by tests aimed at edge cases. Book: Section 8.12.

Common Mistakes

Overwriting the only pointer to something you still need; **reading from a node after leaving it.**

Freeing before unlinking; leaving a pointer to freed memory (`tail`, a returned local address).

Mixing two conventions for an index; moving the scanning pointer after a deletion; forgetting the leftover of a merge.

`x % 2 == 1` **for odd; popping operands in the wrong order; counting instead of remembering.**

Fixed-size buffers; storing the end marker as data; testing only the example.

Book: Section 8.13.

Summary: Five Things to Remember

1. Seven steps: restate with an example, choose the structure, draw, write the invariant, list edge cases, order the pointer changes, test under a memory checker.

2. Before changing a link, keep a way to reach everything you still need.

3. Stacks: Equation (8.1), brackets, shunting-yard with the right operand popped first. Queues: Equation (8.5) with a count; a linked queue resets `tail`.

4. By value copies, by reference renames, by pointer passes an address; Java and Python return the new first node.

5. Lists: stand on the predecessor, unlink then free, three pointers to reverse, a dummy to merge (Equation (8.6)).

Book: Section 8.14.

Next Time

Chapter 9 — Doubly and Circular Linked Lists. Every node also remembers the node before it: back deletion in $O(1)$, a list with a tail, the circular version, and the deque.

Problem 8.2 was the preview: two arrows per node, and both set on every insertion.

Before then: work Exercises 8.3, 8.5, 8.7 and 8.11, and run `wrong/p812_wrong.cpp` under AddressSanitizer yourself.

Book: Section 8.15.

Lecture 9 — Doubly and Circular Linked Lists

Course: Data Structure (EF234201)

Book: Irfan Subakti, *Data Structure with C++, Java and Python*, Chapter 9

Lecturer: Irfan Subakti — yifana@gmail.com

Today every node gets a second arrow, pointing backward, and two limits of Chapter 7 disappear. Book: Chapter 9, Sections 9.1–9.15.

What This Meeting Covers

1. **Two arrows per node** — `prev` and `next`, the two-way invariant, and what the second pointer costs in memory.
2. **The list through `head`, then `head` and `tail`** — every everyday operation, drawn before and after, with its cost.
3. **Head node, circle, sentinel** — three changes of representation, until no special case is left.
4. **Two applications** — a deque for the Pustaka Ceria returns desk, and a “recently borrowed” panel (the LRU pattern).
5. **Costs, measured; memory safety; the libraries** — singly versus doubly linked versus array, Valgrind and AddressSanitizer, `std::list`, `LinkedList`, `collections.deque`.

Every operation is drawn before and after, every cost is derived, and every program is complete and memory-checked. Book: Section 9.1.

Where We Left Off: Two Limits

Going backward. A singly linked node knows only the node after it; to step back we start again at `head` and walk.

Deleting at the back. Section 7.8 made back insertion $O(1)$ with `tail`, but back deletion stayed $\Theta(n)$: the node *before* the last must change, and nobody knows where it is.

The same limit hides in deletion by key: Section 7.6.5 needed a trailing pointer `prev` one step behind `cur`.

The plan: add one field, `prev`, to every node. Walk both ways, delete the last node in $O(1)$, delete *any* held node in $O(1)$.

Chapter 8, Problem 8.2, already walked a two-way list forward and backward; today we develop the idea fully, together with its price: twice the pointer work and 8 more bytes per node. Book: Section 9.1.

The Doubly Linked List

> **DEFINITION.** A *doubly linked list* is a sequence of nodes in which every node stores one element (its *data*) and two pointers: `next`, to the following node, and `prev`, to the preceding node. In a *non-circular* doubly linked list (*DLLNC*) the first node's `prev` and the last node's `next` are null. The list is reached through `head`, often together with `tail`; an empty list has no nodes and null external pointers.

Picture a train: every carriage is coupled to the one in front and the one behind, the two couplings of neighbors agree, and the conductor can walk to the end and back.

Double: two pointer fields per node. *Linked list*: nodes joined by pointers, not side by side. *Non-circular*: both chains end in null.

Removing a carriage changes two couplings, adding one changes four: exactly the pointer counts of deletion and insertion. Book: Sections 9.2.1–9.2.2.

Anatomy of a Doubly Linked List

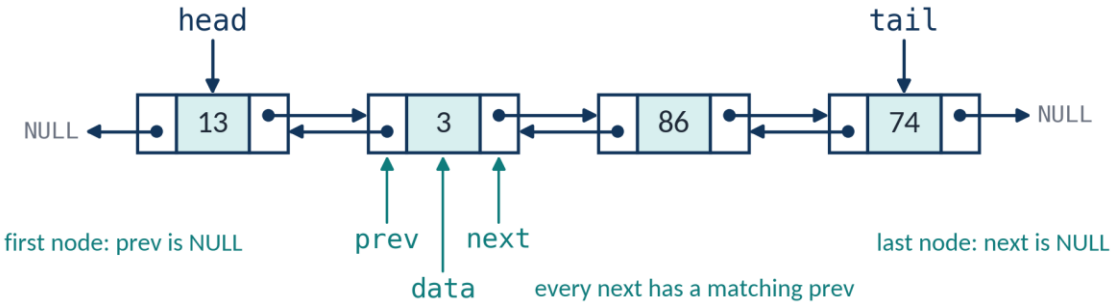


Figure 9.1 A doubly linked list holding 13, 3, 86 and 74. Each node has a `prev` field, a data field and a `next` field; `head` points at the first node and `tail` at the last, and both ends of the chain are NULL.

*In **Figure 9.1** each node has three compartments: `prev`, data, `next`. `next` arrows leave the upper half and run right; `prev` arrows leave the lower half and run left. Between two neighbors there are always two arrows. Book: Section 9.2.2 and Figure 9.1.*

The Two-Way Invariant

DEFINITION. The *two-way invariant* of a non-circular doubly linked list: for every node p, Equation (9.1) holds, together with $head.prev = \text{NIL}$ and $tail.next = \text{NIL}$ when the list is not empty.

$$p.next \neq \text{NIL} \;\Rightarrow\; p.next.prev = p \qquad \text{and} \qquad p.prev \neq \text{NIL} \;\Rightarrow\; p.prev.next = p \tag{9.1}$$

In words: step forward and then backward, and you are back where you started; the same holds backward-then-forward.

Every operation of this chapter *relies* on it without checking it: `backDeletion` trusts `tail->prev`; `deleteNode(p)` trusts that `p->prev->next` is `p`.

If the invariant breaks anywhere, later operations silently corrupt the list further. Section 9.4.5 writes a checker; Section 9.11 shows the damage in memory. Book: Section 9.2.2 and Equation (9.1).

Declaring the Node

The node of Chapter 7 gains exactly one field; a new node starts with both pointers null (**Listing 9.1**).

Listing 9.1 The start of `dll_node.cpp`: the self-referential node type and `makeNode`.

```
// dll_node.cpp -- a doubly linked node: declared, created, and linked by hand.
#include <cstdio>

typedef struct MyNode {
    int data;
    struct MyNode *prev;  // address of the node before, or nullptr at the front
    struct MyNode *next;    // address of the node after, or nullptr at the back
} TNode;

TNode *makeNode(int newData) {
    TNode *newNode = new TNode;
    newNode->data = newData;
    newNode->prev = nullptr;              // a new node is linked to nothing yet
    newNode->next = nullptr;
    return newNode;
}
```

Both `prev` and `next` have the type "pointer to `struct MyNode`". By convention `prev` comes first, matching the left-to-right order of the diagrams. Book: Section 9.2.3.

What the Second Pointer Costs

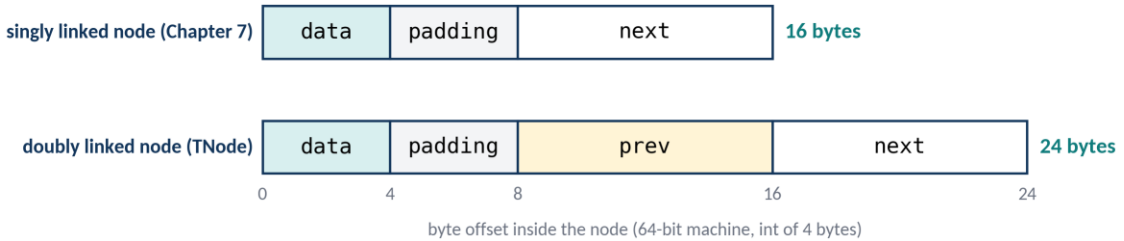


Figure 9.2 The memory layout of a node holding an `int` on a 64-bit machine: 16 bytes for a singly linked node, 24 bytes for a doubly linked one. The padding keeps each pointer on an address divisible by 8.

***Figure 9.2**: the `int` takes bytes 0–3, padding 4–7 keeps each pointer on an address divisible by 8, `prev` is at 8–15 and `next` at 16–23. The program prints `sizeof(TNode) = 24 bytes`. Book: Section 9.2.4 and Figure 9.2.*

The Memory Formula

With data of d bytes, pointers of p bytes and allocator overhead of a bytes per node, Equation (9.2) compares the two lists:

$$M_{\text{DLL}} = n\,(d + 2p + a) \quad \text{bytes, against} \quad M_{\text{SLL}} = n\,(d + p + a). \tag{9.2}$$

The difference is exactly $n\,p$: one pointer per element, 8 MB for a million elements.

For `int` data it is a heavy tax (24 bytes of node for 4 bytes of data); for 200-byte book records it is 4 percent.

The rule of thumb of Chapter 7 applies with more force: doubly linked lists of tiny elements are expensive, of large records they are not. Book: Section 9.2.4 and Equation (9.2).

`frontInsertion`: Three Assignments

Pseudocode 9.1 changes three pointers where the singly linked list changed two: the old first node must now point *back* at the new one.

Pseudocode 9.1 `FRONT-INSERTION` in a doubly linked list reached through `head`: three pointer assignments, and the middle one only when the list was not empty.

```
FRONT-INSERTION(L, x):
 1.  z ← a new node;  z.data ← x;  z.prev ← NIL
 2.  z.next ← L.head              // step 1: the new node points forward at the old first
 3.  if L.head ≠ NIL
 4.      L.head.prev ← z          // step 2: the old first node points back at the new one
 5.  L.head ← z                   // step 3: the new node becomes the first node
```

Line 3 is the first *special case* of the chapter: an empty list has no old first node to update.

Step 3 must come last, because steps 1 and 2 read `head` to find the old first node; steps 1 and 2 may be exchanged. Cost $O(1)$. The functions take `head` by reference, `TNode *&head`, when they may change it. Book: Section 9.3.2.

Front Insertion, Step by Step

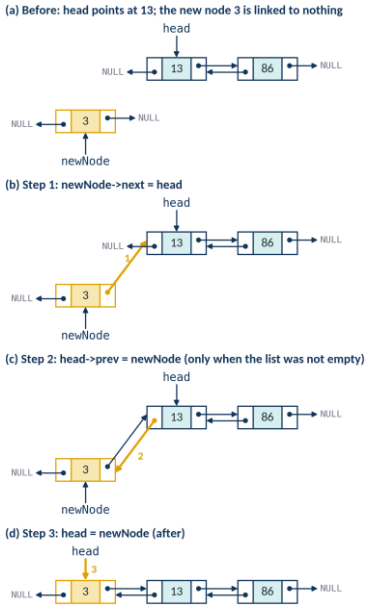


Figure 9.3 `frontInsertion(3)` on the list 13, 86: (a) before, (b) the new node points forward at the old first node, (c) the old first node points back at the new one, (d) `head` moves to the new node.

*In **Figure 9.3**, after step 2 the new node is correctly linked to 13 in both directions, yet the list does not know it until step 3 moves `head`. Book: Section 9.3.2 and Figure 9.3.*

`backInsertion` with Only `head`

Pseudocode 9.2: walk to the last node exactly as in Section 7.5.3, then link in both directions.

Pseudocode 9.2 `BACK-INSERTION` with only a head pointer: walk to the last node, then link in both directions.

```
BACK-INSERTION(L, x):
 1.  z ← a new node;  z.data ← x;  z.prev ← NIL;  z.next ← NIL
 2.  if L.head = NIL              // empty list: the new node is also the first
 3.      L.head ← z
 4.  else
 5.      aux ← L.head
 6.      while aux.next ≠ NIL
 7.          aux ← aux.next
 8.      aux.next ← z             // step 1: the old last node points forward at z
 9.      z.prev ← aux             // step 2: z points back at the old last node
```

Line 9 is the only new line: without it the forward walk shows 74, but a backward walk starting at 74 stops at once. Cost $\Theta(n)$ for the walk: only a pointer to the back would shorten it. Book: Section 9.3.3.

Back Insertion, Drawn

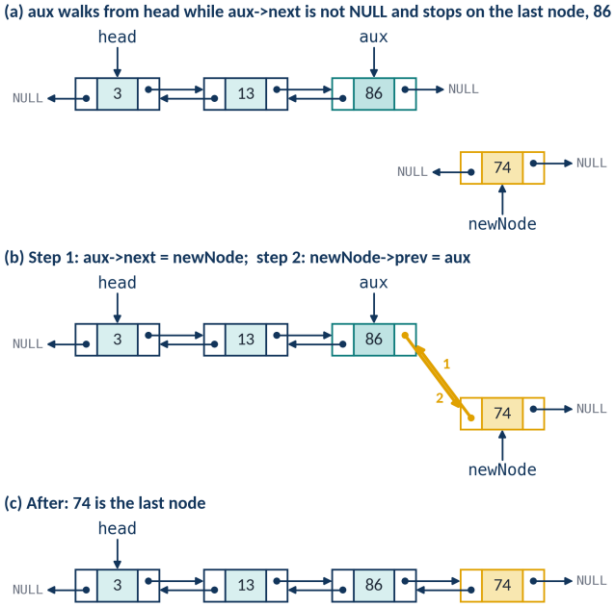


Figure 9.4 `backInsertion(74)` on the list 3, 13, 86: (a) `aux` stops on the last node, (b) two links are set, one in each direction, (c) after.

Figure 9.4 *traces the insertion of 74 into 3, 13, 86; panel (a) is the walk of Chapter 7. Book: Section 9.3.3 and Figure 9.4.*

Traversing Backward

Pseudocode 9.3: phase 1 walks forward to the last node, phase 2 follows `prev` to the front.

Pseudocode 9.3 `SHOW-BACKWARD` with only a head pointer: walk to the last node, then visit every node from the last to the first.

```
SHOW-BACKWARD(L):
 1.  if L.head = NIL
 2.      return
 3.  aux ← L.head
 4.  while aux.next ≠ NIL        // phase 1: walk forward to the last node
 5.      aux ← aux.next
 6.  while aux ≠ NIL             // phase 2: visit every node, following prev
 7.      visit aux
 8.      aux ← aux.prev
```

Phase 1 tests `aux.next` (it must *stop on* the last node); phase 2 tests `aux` (it must *visit* every node).

About $2n$ steps and no extra memory. A singly linked list needed $\Theta(n^2)$ time or a $\Theta(n)$ stack to print backward. With `tail`, phase 1 disappears. Book: Section 9.3.4.

Both Traversals, Drawn

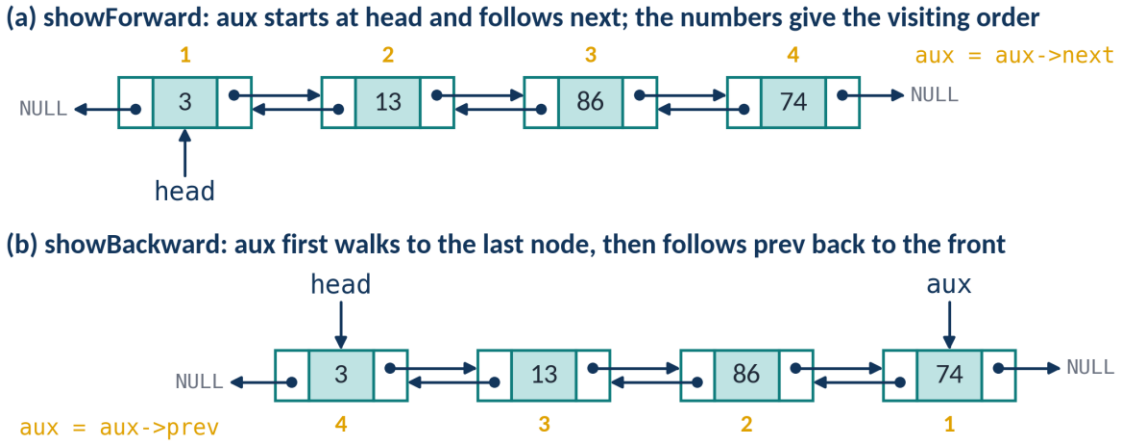


Figure 9.5 Traversing the list 3, 13, 86, 74: (a) forward from `head`, following `next`; (b) backward, following `prev` from the last node. The numbers give the order of the visits.

*The numbers in **Figure 9.5** give the order of the visits in each direction. Book: Section 9.3.4 and Figure 9.5.*

`frontDeletion`: Clear the New `prev`

Pseudocode 9.4 is save, move, delete from Section 7.5.5, plus one assignment.

Pseudocode 9.4 `FRONT-DELETION`: remember the first node, move `head`, clear the new first node's `prev`, release the old node.

```
FRONT-DELETION(L):
 1.  if L.head = NIL
 2.      report "the list is empty" and return
 3.  del ← L.head                 // remember the first node
 4.  L.head ← L.head.next         // step 1: the second node becomes the first
 5.  if L.head ≠ NIL
 6.      L.head.prev ← NIL        // step 2: the new first node has no predecessor
 7.  free del
```

TRAP. Line 6 is the line a programmer coming from singly linked lists forgets. The list still prints forward, but 13's `prev` holds the address of a released node: a dangling pointer *inside* the list.

Line 5 is again a special case: a one-node list has no new first node to update. Cost $O(1)$. Book: Section 9.3.5.

Front Deletion, Drawn

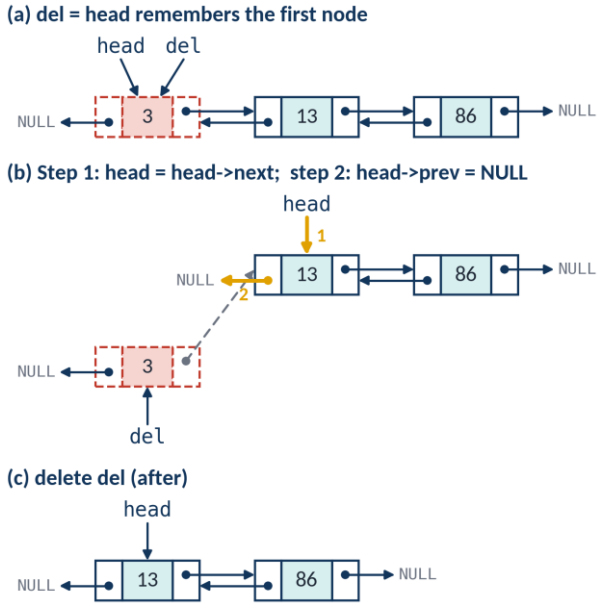


Figure 9.6 `frontDeletion()` on the list 3, 13, 86: (a) `del` remembers the first node, (b) `head` moves to 13 and 13's `prev` becomes NULL, (c) the old first node is deleted.

***Figure 9.6**: `del` remembers 3, `head` moves to 13 and 13's `prev` becomes NULL, then 3 is released. Book: Section 9.3.5 and Figure 9.6.*

`backDeletion` with Only `head`

Pseudocode 9.5: the helper walks to the last node itself; its predecessor is simply `del.prev`. No more awkward `aux.next.next`.

Pseudocode 9.5 `BACK-DELETION` with only a head pointer: walk to the last node; its predecessor is `del.prev`.

```
BACK-DELETION(L):
 1.  if L.head = NIL
 2.      report "the list is empty" and return
 3.  del ← L.head
 4.  while del.next ≠ NIL        // walk to the LAST node
 5.      del ← del.next
 6.  if del.prev = NIL           // it is also the first: the only node
 7.      L.head ← NIL
 8.  else
 9.      del.prev.next ← NIL     // its predecessor becomes the last node
10.  free del
```

The one-node case is recognized by `del.prev = NIL` after the walk. The code is simpler, but the walk is still $\Theta(n)$: the only way in is at the front. Book: Section 9.3.6.

Back Deletion, Drawn

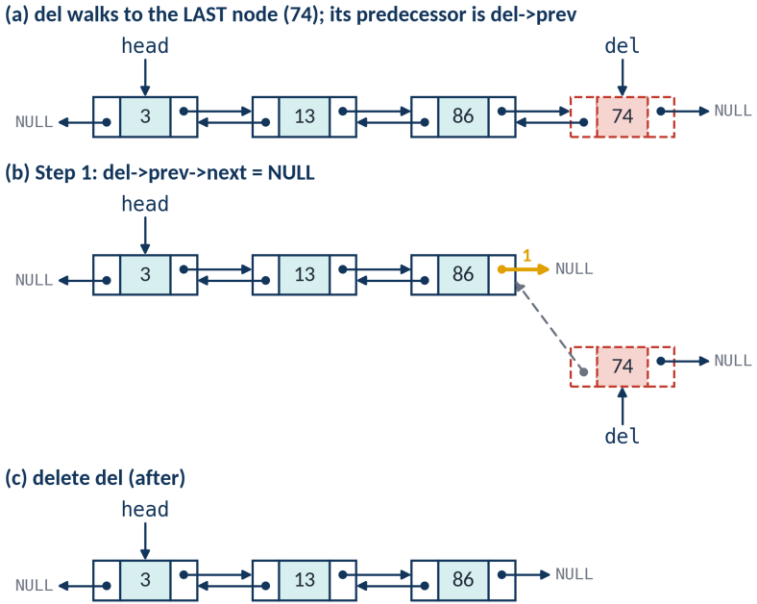


Figure 9.7 `backDeletion()` with only `head` on the list 3, 13, 86, 74: (a) `del` walks to the last node, (b) its predecessor, `del->prev`, stops pointing at it, (c) the node is deleted.

***Figure 9.7**: `del` walks to 74, its predecessor 86 stops pointing at it, and the node is deleted. Book: Section 9.3.6 and Figure 9.7.*

The Same List in Python

Listing 9.2: no `delete`, yet `front_deletion` still clears the new first node's `prev`.

Listing 9.2 `front_deletion` from `dll_head.py`.

```
    def front_deletion(self):
        if self.is_empty():
            print("The linked list is empty")
            return
        d = self.head.data
        self.head = self.head.next  # the old first node becomes garbage
        if self.head is not None:
            self.head.prev = None  # and nothing may point back at it
        print(f"{d} has been deleted by frontDeletion()")
```

A reference can never point at freed memory, but a stale backward link breaks the invariant *and* keeps the old node alive: a logic error and a memory leak at once.

`DllHead.java` does the same with `head.prev = null`; both print exactly the output of the C++ program. Book: Section 9.3.9.

Head and Tail: Back Deletion in $O(1)$

Pseudocode 9.6: the node before the last is `tail.prev`, so there is no walk at all.

Pseudocode 9.6 `BACK-DELETION` with head and tail: the node before the last is `tail.prev`, so no walk is needed.

```
BACK-DELETION-WITH-TAIL(L):
 1.  if L.tail = NIL
 2.      report "the list is empty" and return
 3.  del ← L.tail
 4.  L.tail ← L.tail.prev        // step 1: no walk: the new last node is tail.prev
 5.  if L.tail ≠ NIL
 6.      L.tail.next ← NIL       // step 2: the new last node points at nothing
 7.  else
 8.      L.head ← NIL            // the list had one node and is now empty
 9.  free del
```

Line 4 is the line Chapter 7 could not write. Back deletion is the mirror image of front deletion: swap `head`/`tail` and `next`/`prev`. `head` and `tail` are now both null or both non-null. Book: Section 9.4.1.

Back Deletion with a Tail, Drawn

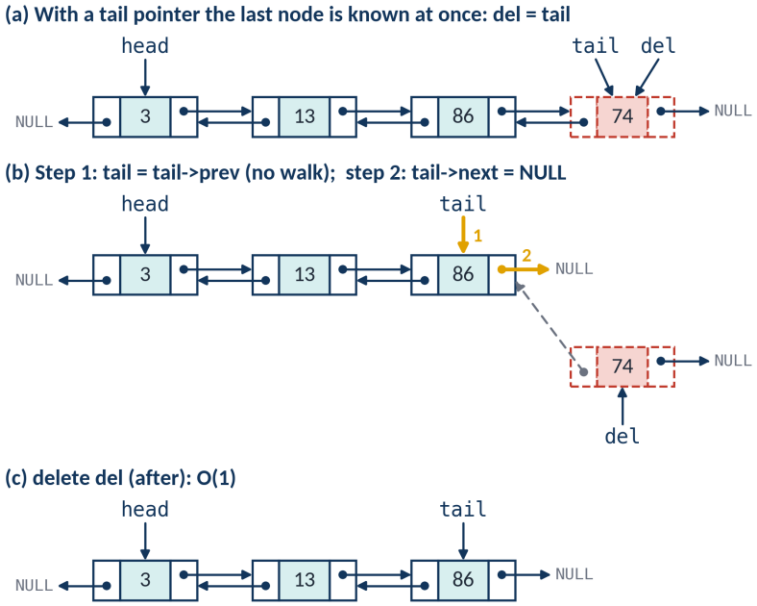


Figure 9.8 Back deletion with a tail pointer on 3, 13, 86, 74: (a) `del` = `tail`, (b) `tail` moves back along `prev` and the new last node's `next` becomes NULL, (c) after: $O(1)$, with no walk.

Figure 9.8: *`tail` moves one step back along `prev`, 86 becomes the last node, 74 is released, in a constant number of steps whatever the length. Book: Section 9.4.1 and Figure 9.8.*

`insertAfter`: Four Arrows

Between `p`, the new node `n` and the successor s there are four arrows: $p \to n$, $n \to s$ forward; $s \to n$, $n \to p$ backward (**Pseudocode 9.7**).

Pseudocode 9.7 `INSERT-AFTER`: link a new node after node `p` with four pointer assignments, in $O(1)$ time; step 4 must come after steps 2 and 3.

```
INSERT-AFTER(L, p, x):
 1.  n ← a new node;  n.data ← x
 2.  n.prev ← p                  // step 1
 3.  n.next ← p.next             // step 2
 4.  if p.next = NIL             // p was the last node
 5.      L.tail ← n
 6.  else
 7.      p.next.prev ← n         // step 3: the successor points back at n
 8.  p.next ← n                  // step 4: p points forward at n
```

> **SAFE RULE.** First set the new node's own two pointers, then update the neighbors, and update the link you used to find a neighbor *last*.

Lines 3 and 7 read `p.next`; line 8 overwrites it. If line 8 ran first, the new node would point at itself and 50 would be lost. `insertBefore` is the mirror image, impossible in $O(1)$ for a singly linked list. Book: Section 9.4.2.

Insert After, Step by Step

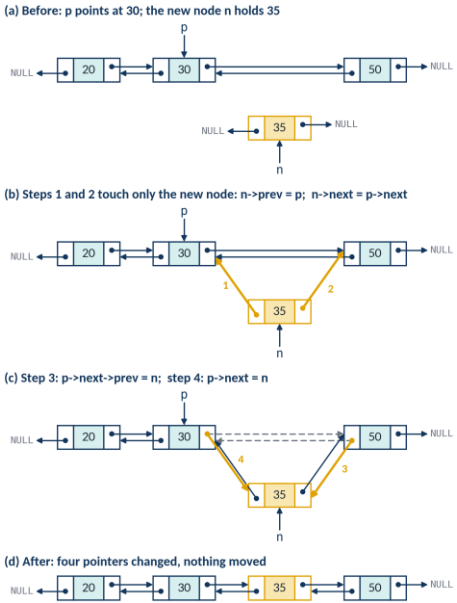


Figure 9.9 `insertAfter(p, 35)` with `p` at 30: (a) before, (b) the new node points at both neighbors, (c) both neighbors point at the new node, (d) after.

Figure 9.9*: steps 1 and 2 touch only the new node; steps 3 and 4 redirect the neighbors; grey dashed arrows are the links replaced. Book: Section 9.4.2 and Figure 9.9.*

Sorted Insertion, Drawn

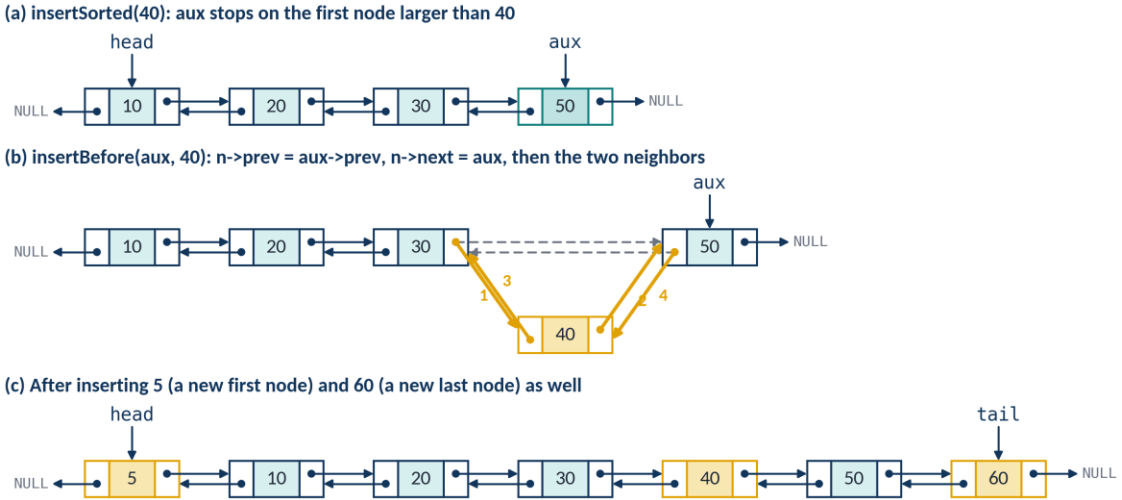


Figure 9.10 `insertSorted`: (a) for 40, `aux` stops on 50, the first larger value, (b) the new node is linked in before `aux` in four steps, (c) the list after 5 and 60 are also inserted, at the front and at the back.

***Figure 9.10**: for 40, `aux` stops on 50; then 5 goes to the front and 60 to the back, the three cases of the procedure. Book: Section 9.4.3 and Figure 9.10.*

`deleteNode`: Two Assignments

Pseudocode 9.8: the predecessor is `p.prev`, the successor `p.next`; make them skip `p`.

Pseudocode 9.8 `DELETE-NODE`: unlink node `p` in $O(1)$ time; no search and no trailing pointer are needed.

```
DELETE-NODE(L, p):
 1.  if p.prev = NIL              // p is the first node
 2.      L.head ← p.next
 3.  else
 4.      p.prev.next ← p.next     // step 1: the predecessor skips p going forward
 5.  if p.next = NIL              // p is the last node
 6.      L.tail ← p.prev
 7.  else
 8.      p.next.prev ← p.prev     // step 2: the successor skips p going backward
 9.  free p
```

`frontDeletion` is `DELETE-NODE(L, L.head)`; `backDeletion` is `DELETE-NODE(L, L.tail)`; `deleteKey` is a search followed by `DELETE-NODE`.

When the pointer comes from a hash map, an iterator or the node itself, a singly linked list must search for the predecessor, $\Theta(n)$; a doubly linked list does not, $O(1)$. Book: Section 9.4.4.

Deletion by Key, Drawn

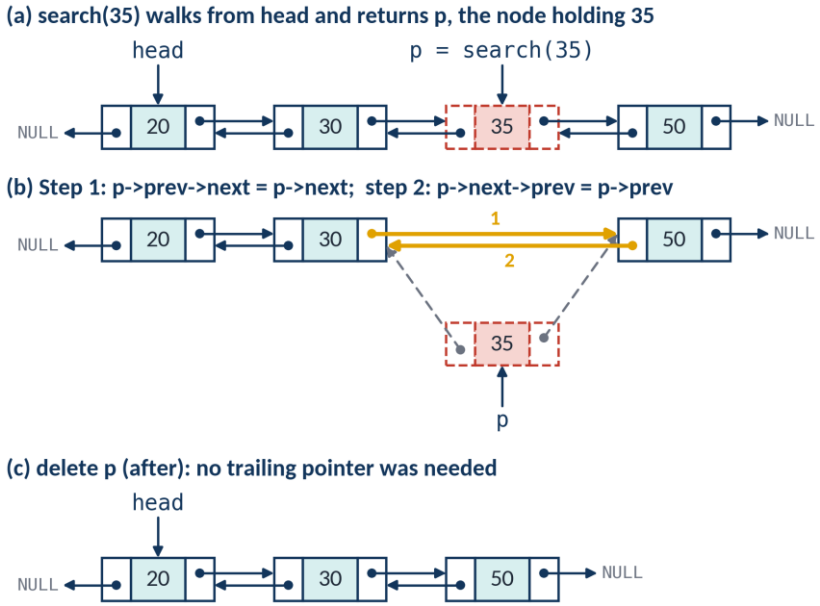


Figure 9.11 `deleteKey(35)`: (a) `search` returns `p`, (b) the predecessor and the successor are linked to each other, (c) `p` is deleted.

*In **Figure 9.11**, `p`'s own pointers are read but never changed: the node is abandoned by its neighbors and then freed. Book: Section 9.4.4 and Figure 9.11.*

`deleteNode` in C++

Listing 9.3 `deleteNode`, the two end deletions and `deleteKey` from `dll_ops.cpp`.

```
// unlink p in O(1): no walk, no trailing ptr
void deleteNode(DList &L, Node *p) {
    if (p->prev == nullptr) L.head = p->next;         // p was the first node
    else p->prev->next = p->next;                     // 1. predecessor skips p
    if (p->next == nullptr) L.tail = p->prev;         // p was the last node
    else p->next->prev = p->prev;  // 2. successor skips p back
    delete p;
}

void frontDeletion(DList &L) { if (!isEmpty(L)) deleteNode(L, L.head); }
void backDeletion(DList &L) { if (!isEmpty(L)) deleteNode(L, L.tail); }

bool deleteKey(DList &L, int key) {
    Node *p = search(L, key);
    if (p == nullptr) return false;
    deleteNode(L, p);
    return true;
}
```

Listing 9.3: one procedure with two tests covers every deletion in the list.

`checkLinks` *in the same program walks the list once and verifies Equation (9.1); calling it after every operation in a test catches a forgotten backward assignment at once. Book: Sections 9.4.4–9.4.5.*

Worked Example: Tracing `dll_ops.cpp`

```
start:                    10 20 30 50
insertAfter(30, 35):      10 20 30 35 50
insertBefore(10, 5):      5 10 20 30 35 50
insertSorted 40, 1, 60:   1 5 10 20 30 35 40 50 60
backward:                 60 50 40 35 30 20 10 5 1
links consistent:         yes
search(35):               30 <- 35 -> 40
deleteKey(35):            1 5 10 20 30 40 50 60
front and back deletion:  5 10 20 30 40 50
```

`insertBefore(10, 5)` changes `head`; the sorted insertions hit the middle (40), the front (1) and the back (60).

The backward line is the forward line reversed, and `checkLinks` agrees; the three deletions remove a middle node, the front and the back, each in $O(1)$ once the node is known. Book: Section 9.4.6.

Costs with Head and Tail

Table 9.1 Costs with head and tail pointers, for a list of n nodes. The doubly linked list turns every operation at a known position into $O(1)$, at the price of twice the pointer work.

Operation	Singly linked, head and tail (Chapter 7)	Doubly linked, head and tail
front insertion, front deletion	$O(1)$	$O(1)$
back insertion	$O(1)$	$O(1)$
back deletion	$\Theta(n)$	$O(1)$ (**Pseudocode 9.6**)
insert after a held node	$O(1)$	$O(1)$ (**Pseudocode 9.7**)
insert before a held node	$\Theta(n)$	$O(1)$
delete a held node	$\Theta(n)$	$O(1)$ (**Pseudocode 9.8**)
search, delete by key, sorted insertion	$\Theta(n)$	$\Theta(n)$
forward traversal	$\Theta(n)$	$\Theta(n)$
backward traversal	$\Theta(n^2)$, or $\Theta(n)$ with a stack	$\Theta(n)$
pointer assignments per insertion / deletion	2 / 1	4 / 2

***Table 9.1**: every operation at a known position becomes $O(1)$, at the price of twice the pointer work. Book: Section 9.4.6.*

A Head Node Removes the Front Cases

Every special case so far has one cause: a node at an *end* has a null pointer on one side, so the operation must write to `head` or `tail` instead.

> **DEFINITION.** A doubly linked list *with a head node* keeps one dataless node permanently at the front; `head` always points at it, the first real node is `head->next`, and the empty list is the head node alone.

Front insertion becomes `insertAfter(head, x)`; `deleteNode(p)` never tests `p->prev`. One test remains, at the back.

In the empty list `tail` points at the head node, so `backInsertion(x)` is `insertAfter(tail, x)`. `destroy` must release the head node too; forgetting it is a real leak. Book: Sections 9.5.1–9.5.2.

The Head Node, Drawn

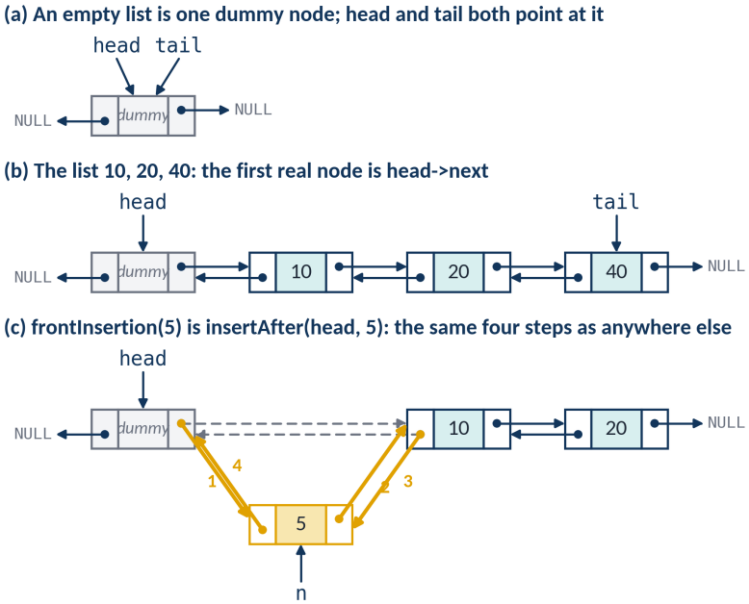


Figure 9.12 A doubly linked list with a head node: (a) the empty list is one dummy node, and `tail` points at it too, (b) the list 10, 20, 40, (c) front insertion is `insertAfter(head, 5)`, with the same four steps as any insertion.

***Figure 9.12**: because every real node now has a predecessor, no operation ever writes to `head` again. Book: Section 9.5.1 and Figure 9.12.*

Closing Both Chains: the Circle

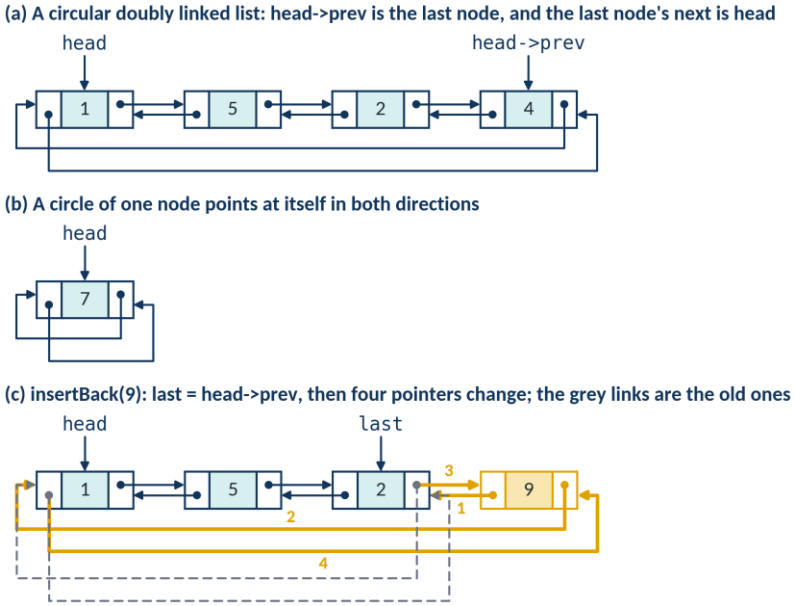


Figure 9.13 A circular doubly linked list reached through `head`: (a) four nodes; the last node is `head->prev`, (b) a circle of one node, (c) `insertBack(9)` links the new node between the last node and `head`.

***Figure 9.13**: the last node's `next` is the first node and the first node's `prev` is the last. No `tail`, yet the last node is `head->prev`: one pointer gives $O(1)$ access to both ends. Book: Section 9.6.1 and Figure 9.13.*

A New Stop Signal

There is no null to wait for; stop on returning to the start, and visit the start too, so use `do-while`:

```
const Node *p = head;
do {
    visit(p);
    p = p->next;          // backward: start at head->prev and follow prev
} while (p != head);
```

Test for the empty list *before* the loop. Deletion without a sentinel keeps two special cases: the only node, and the first node.

The lobby carousel of Pustaka Ceria (`cdll.cpp`) is this list with a cursor: Next and Previous wrap around, past the seam, without a single test. Book: Sections 9.6.2–9.6.3.

The Sentinel

> **DEFINITION.** A *circular doubly linked list with a sentinel* keeps one dataless node `nil` permanently in the circle. `nil.next` is the first node and `nil.prev` the last. The empty list is `nil` alone, with `nil.next = nil.prev = nil`.

No pointer is ever null: every node, including `nil`, sits in the circle.

No external pointer ever changes: the first and last nodes are read as `nil.next` and `nil.prev`.

This is the representation of Cormen et al. in Introduction to Algorithms, and the one inside most production linked lists. Book: Sections 9.7.1–9.7.2.

The Sentinel, Drawn

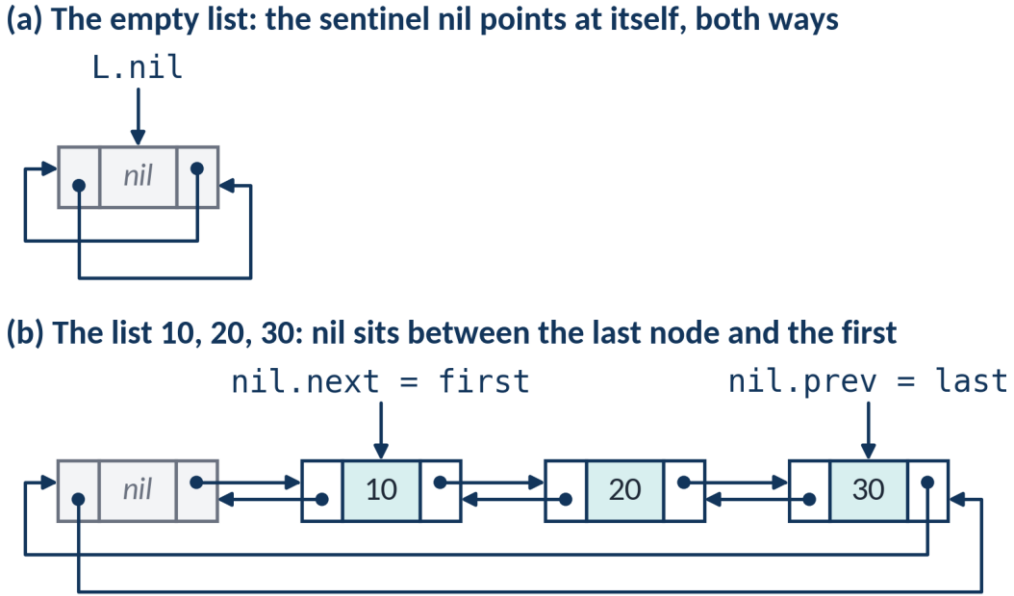


Figure 9.14 A circular doubly linked list with a sentinel: (a) the empty list is the sentinel alone, pointing at itself both ways, (b) the list 10, 20, 30; `nil.next` is the first node and `nil.prev` the last.

Figure 9.14*: the head node of Section 9.5 and the circle of Section 9.6, combined. Book: Section 9.7.1 and Figure 9.14.*

Insertion and Deletion Without an `if`

Pseudocode 9.9 `LIST-INSERT-AFTER` in a circular list with a sentinel: the four steps of **Pseudocode 9.7** without its test.

```
LIST-INSERT-AFTER(p, x):     // p is a node of the list, or nil
 1.  n ← a new node;  n.data ← x
 2.  n.prev ← p                  // step 1
 3.  n.next ← p.next             // step 2
 4.  p.next.prev ← n             // step 3: never NIL; at worst it is nil
 5.  p.next ← n                  // step 4
```

Pseudocode 9.10 `LIST-DELETE` in a circular list with a sentinel: two assignments, no tests.

```
LIST-DELETE(p):              // p is a real node, never nil
 1.  p.prev.next ← p.next
 2.  p.next.prev ← p.prev
 3.  free p
```

Pseudocode 9.9 *has the four steps of* ***Pseudocode 9.7*** *without its test;* ***Pseudocode 9.10*** *is two assignments. Book: Section 9.7.2.*

Worked Example: the Two Dangerous Cases

Insert into the empty list, $p = nil$: line 2 sets $n.prev = nil$; line 3 sets $n.next = nil$; line 4 sets $nil.prev = n$; line 5 sets $nil.next = n$. A correct one-element list.

Delete the only node, $p.prev = p.next = nil$: line 1 sets $nil.next = nil$; line 2 sets $nil.prev = nil$. The empty list again.

> **WHY IT WORKS.** A special case is a place where the structure is not uniform. A sentinel makes it uniform: every real node has two real neighbors.

Work both cases on the board with the lines of the two procedures; the next page shows the pictures. Book: Section 9.7.2.

The Two Cases, Drawn

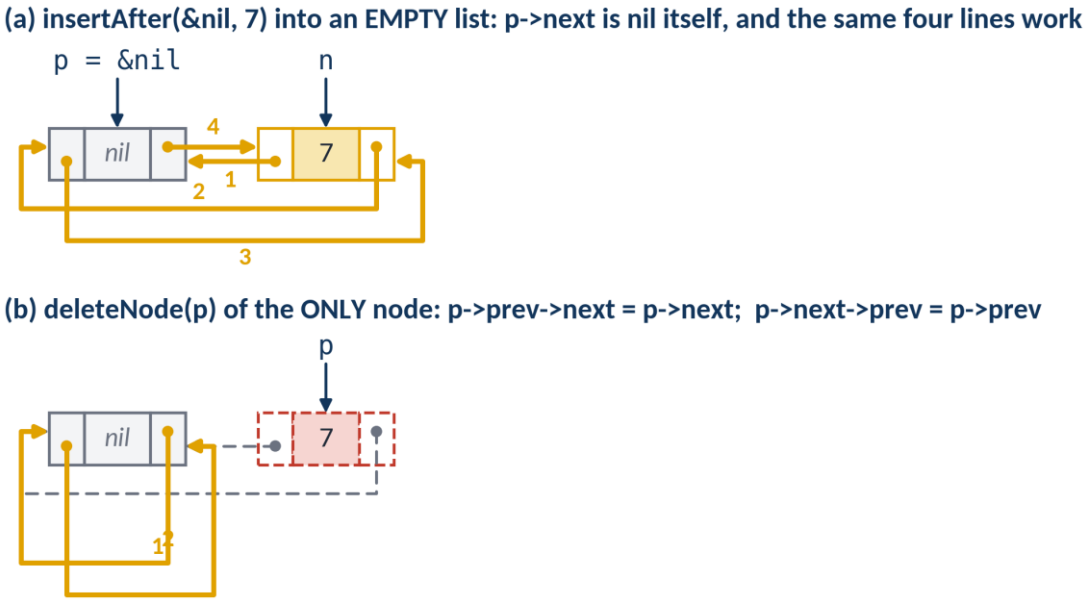


Figure 9.15 The two cases that needed special code before, handled by the ordinary code: (a) inserting after the sentinel into an empty list, (b) deleting the only node.

Figure 9.15*: the cases that needed special code before are handled by the ordinary code. Book: Section 9.7.2 and Figure 9.15.*

Searching with the Key in the Sentinel

Pseudocode 9.11 `LIST-SEARCH-SENTINEL`: store the key in the sentinel, so that the loop needs only one test per step.

```
LIST-SEARCH-SENTINEL(L, key):
 1.  nil.data ← key              // the search is now certain to stop
 2.  p ← nil.next
 3.  while p.data ≠ key
 4.      p ← p.next
 5.  if p = nil
 6.      return NIL              // it stopped on the sentinel: key is absent
 7.  return p
```

$$C_{\text{ordinary}}(n) = 2n + 1, \qquad C_{\text{sentinel}}(n) = n + 1 \quad \text{tests in total.} \tag{9.3}$$

Pseudocode 9.11 cannot run past the end, so the end test leaves the loop; Equation (9.3) counts the tests of an unsuccessful search.

Still $\Theta(n)$ steps; the sentinel halves the constant factor. Knuth analyzes the same trick for arrays, with the key placed just past the last element. Book: Section 9.7.3 and Equation (9.4).

The Sentinel List in C++

Listing 9.4 From the class `SentinelList` in `cdll_sentinel.cpp`: `insertAfter` and `deleteNode` contain no `if`, and the end operations are one-liners.

```
    // p is a node of the list or the sentinel
    Node *insertAfter(Node *p, int x) {
        // 1. n->prev = p   2. n->next = p->next
        Node *n = new Node{x, p, p->next};
        p->next->prev = n;  // 3. never NULL: at worst it is the sentinel
        p->next = n;                        // 4.
        return n;
    }
    void deleteNode(Node *p) {  // p is a real node, never the sentinel
        p->prev->next = p->next;
        p->next->prev = p->prev;
        delete p;
    }

    void frontInsertion(int x) { insertAfter(&nil_, x); }
    void backInsertion(int x) { insertAfter(nil_.prev, x); }
    void frontDeletion() { if (!isEmpty()) deleteNode(nil_.next); }
    void backDeletion() { if (!isEmpty()) deleteNode(nil_.prev); }
```

***Listing 9.4**: the sentinel `nil_` is a member of the object, so it can never leak; copying is forbidden, because a copied sentinel's neighbors would point back at the original. The last line of its run prints `nil points at itself: yes`. Book: Section 9.7.4.*

The Special Cases, Counted

Table 9.2 Special-case tests in the two basic operations, by representation. Only the sentinel removes all of them.

Representation	Tests in insert-after	Tests in delete-node	What the tests guard
non-circular, `head` and `tail`, no head node (Section 9.4)	1	2	the back end on insertion; both ends on deletion
non-circular with a head node (Section 9.5)	1	1	the back end only
circular, through `head`, no sentinel (Section 9.6)	1 (empty list)	2 (only node, first node)	the empty list; the moving `head`
circular with a sentinel (Section 9.7)	0	0	nothing

***Table 9.2**: only the sentinel removes all of them, for the price of one node. Book: Section 9.7.5.*

A Deque for the Returns Desk

DEFINITION. A *deque* (double-ended queue) supports `pushFront`, `pushBack`, `popFront`, `popBack`, `front`, `back`, `isEmpty` and `size`. Used only through `pushFront`/`popFront` it is a stack; through `pushBack`/`popFront`, a queue.

Returned books go to the back of the cart; a reserved book jumps to the **front**; a mistaken scan is taken off the **back**.

On a sentinel list every deque operation is $O(1)$ by Equation (9.4), with no capacity limit:

$$\begin{aligned} \text{FRONT-INSERTION}(x) &= \text{LIST-INSERT-AFTER}(nil, x), \\ \text{BACK-INSERTION}(x) &= \text{LIST-INSERT-AFTER}(nil.prev, x), \\ \text{FRONT-DELETION}() &= \text{LIST-DELETE}(nil.next), \\ \text{BACK-DELETION}() &= \text{LIST-DELETE}(nil.prev). \end{aligned} \tag{9.4}$$

The deletions require a non-empty list, a precondition rather than a special case: `deque.cpp` throws `std::underflow_error` rather than unlink its own sentinel. Book: Sections 9.7.2 and 9.8.1–9.8.2.

The Returns Cart, Drawn

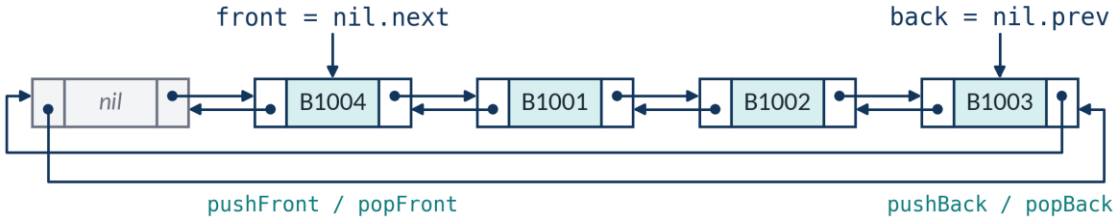


Figure 9.16 The returns desk as a deque on a circular doubly linked list with a sentinel: the front is `nil.next`, the back is `nil.prev`, and all four operations are $O(1)$.

***Figure 9.16**: the front is `nil.next`, the back is `nil.prev`. Book: Section 9.8.2 and Figure 9.16.*

“Recently Borrowed”: the LRU Pattern

The kiosk panel shows the $k = 4$ most recently borrowed *distinct* titles, newest first.

A title borrowed again moves to the top; when a new title overflows the panel, the **least recently used** falls off.

Two structures work together: a **sentinel list** keeps the order of recency, a **hash map** maps each title to *the address of its node*.

The same policy runs browser caches, database page caches and operating-system page tables. The map hands us a node in the middle; only a doubly linked node leaves the middle without a search. Book: Sections 9.9.1–9.9.2.

The Panel and Its Map

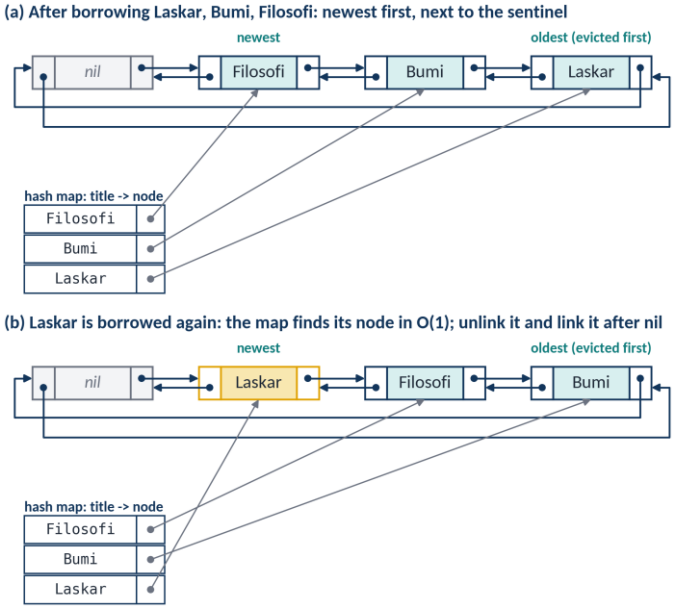


Figure 9.17 The "recently borrowed" panel: a circular doubly linked list with a sentinel keeps the titles from newest to oldest, and a hash map points from each title to its node. (a) After three borrowings; (b) after *Laskar* is borrowed again and moved to the front; the map itself does not change.

***Figure 9.17**: the map finds the node; the list moves it. Book: Section 9.9.2 and Figure 9.17.*

BORROW in $O(1)$ Expected Time

Pseudocode 9.12 BORROW on the "recently borrowed" panel: a hash-map lookup plus a constant number of pointer assignments.

```
BORROW(title):
 1.  p ← map.find(title)
 2.  if p ≠ NIL                  // already on the panel
 3.      unlink p                // two assignments (LIST-DELETE without free)
 4.      link p after nil        // four assignments: p is now the newest
 5.      return
 6.  n ← a new node for title;  link n after nil;  map[title] ← n
 7.  if map.size > k             // the panel overflowed
 8.      old ← nil.prev          // the least recently borrowed title
 9.      unlink old;  map.erase(old.title);  free old
```

$$T_{\text{BORROW}} = O(1) \text{ expected, whatever } k \text{ is,} \tag{9.5}$$

Pseudocode 9.12 *is a map lookup plus a constant number of pointer assignments, hence Equation (9.5). An array or a singly linked list would pay a $\Theta(k)$ search every time. Book: Section 9.9.2 and Equation (9.5).*

`borrow` in C++

Listing 9.5 The method `borrow` of `recently_borrowed.cpp`: move to the front, or insert and evict the oldest.

```
    std::string borrow(const std::string &title) {
        auto it = where_.find(title);
        if (it != where_.end()) {  // already on the panel: move it to the front
            unlink(it->second);
            linkFront(it->second);
            return "";
        }
        Node *n = new Node{title, nullptr, nullptr};
        linkFront(n);
        where_[title] = n;
        if (where_.size() <= capacity_) return "";
        Node *oldest = nil_.prev;  // over capacity: evict the least recent
        std::string gone = oldest->title;
        unlink(oldest);
        where_.erase(gone);
        delete oldest;
        return gone;
```

***Listing 9.5**: `unlink` and `linkFront` are the two sentinel procedures; the node is moved without being freed. Book: Section 9.9.3.*

Worked Example: Eight Borrowings, $k = 4$

```
borrow Laskar Pelangi
borrow Bumi Manusia
borrow Filosofi Kopi
borrow Laskar Pelangi
   panel: [Laskar Pelangi] [Filosofi Kopi] [Bumi Manusia]
borrow Cantik Itu Luka
borrow Ronggeng Dukuh Paruk  (drops off: Bumi Manusia)
borrow Filosofi Kopi
borrow Negeri 5 Menara  (drops off: Laskar Pelangi)
   panel: [Negeri 5 Menara] [Filosofi Kopi] [Ronggeng Dukuh Paruk] [Cantik Itu Luka]
```

Trace it on the board: Laskar moves from bottom to top without dropping anything; Bumi Manusia, untouched since the second borrowing, is the first to fall; at the end Laskar has become the least recent. Book: Section 9.9.3.

Array, Singly Linked, Doubly Linked

Table 9.3 The dynamic array, the singly linked list and the doubly linked list compared, for n elements (selected rows).

Operation	Dynamic array	Singly linked (head, tail)	Doubly linked (head, tail or sentinel)
access the element at index k	$O(1)$	$\Theta(k)$	$\Theta(\min(k, n-1-k))$
insert at the front	$\Theta(n)$	$O(1)$	$O(1)$
delete at the back	$O(1)$	$\Theta(n)$	$O(1)$
insert before a held position	$\Theta(n)$ (shift)	$\Theta(n)$	$O(1)$
delete a held position	$\Theta(n)$ (shift)	$\Theta(n)$	$O(1)$
search by value, sorted	$\Theta(\lg n)$ (binary search)	$\Theta(n)$	$\Theta(n)$
traverse backward	$\Theta(n)$	$\Theta(n^2)$, or $\Theta(n)$ with a stack	$\Theta(n)$
extra memory per element	none (plus spare capacity)	one pointer	two pointers

***Table 9.3**: "a held position" means the program already has a pointer, iterator or index. From the nearer end, index k costs $\min(k, n-1-k)$ steps, about $n/4$ on average: still $\Theta(n)$. Book: Section 9.10.1 and Table 9.4.*

Measuring What `prev` Buys

Emptying a singly linked list from the back walks Equation (9.6) steps; the doubly linked list walks none.

$$\sum_{m=2}^{n} (m-2) = 0 + 1 + \cdots + (n-2) = \frac{(n-1)(n-2)}{2}, \tag{9.6}$$

n	SLL back del	DLL back del	SLL by ptr	DLL by ptr
2500	8.68 ms	0.02 ms	2.97 ms	0.01 ms
5000	36.40 ms	0.04 ms	12.38 ms	0.02 ms
10000	156.41 ms	0.08 ms	49.73 ms	0.04 ms
20000	592.91 ms	0.17 ms	232.24 ms	0.08 ms

Each doubling of n multiplies the singly linked times by about 4 (quadratic) and the doubly linked times by about 2 (linear): at 20,000 nodes roughly 3,500 times faster. Book: Section 9.10.2 and Equation (9.7).

The Measurements, Plotted

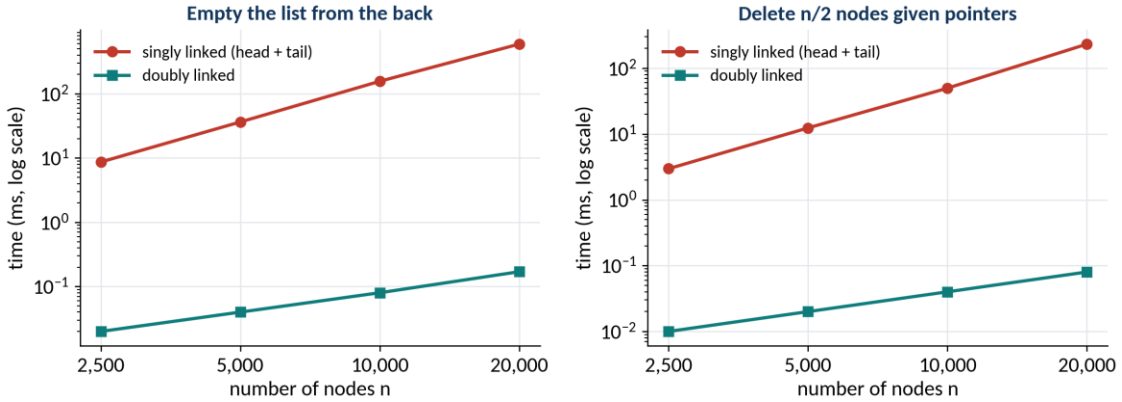


Figure 9.18 The measurements on logarithmic axes: the singly linked list's time grows fourfold per doubling of n (quadratic), the doubly linked list's twofold (linear).

***Figure 9.18**: the timings depend on the machine; the shapes do not. When positions are not held, a search dominates and an array usually wins. Book: Section 9.10.2 and Figure 9.18.*

Three Two-Way-Link Bugs

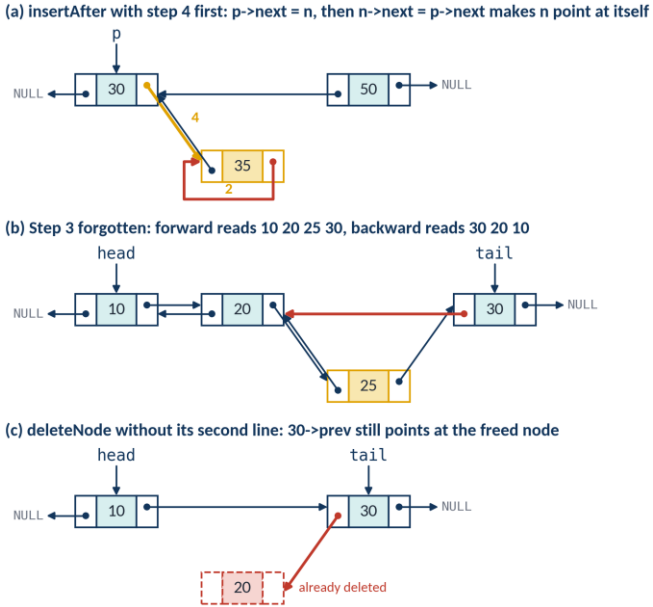


Figure 9.19 Three two-way-link bugs: (a) `insertAfter` with its steps in the wrong order makes the new node point at itself, (b) a forgotten backward link makes the list read differently in the two directions, (c) a forgotten backward link in a deletion leaves a pointer into freed memory.

***Figure 9.19**: each bug updates one direction and not the other. Book: Section 9.11.1 and Figure 9.19.*

What the Tools Say

`halflink`: forward 10 20 25 30, backward 30 20 10, and *no report*: every access was to live memory. Memory checkers find memory errors, not broken invariants.

`stale`: Valgrind reports an *invalid read of size 4* inside a 24-byte block freed by `deleteNodeWrong`; AddressSanitizer reports a *heap-use-after-free* and stops.

`leak`: one block of 24 bytes *definitely lost*: the forgotten heap-allocated sentinel.

Checklist: four on insert, two on delete; every forward write has a backward twin; set both external pointers; release the sentinel; print both ways.

*The second invalid read is of size 8 at offset 8, the `prev` field, exactly where **Figure 9.2** placed it. A correct program ends with "All heap blocks were freed". Book: Sections 9.11.2–9.11.3.*

Doubly Linked Lists in the Libraries

Table 9.4 Doubly linked structures in the C++, Java and Python standard libraries.

	C++ `std::list`	Java `LinkedList`	Java `LinkedHashMap`	Python `collections.deque`	Python `OrderedDict`, `lru_cache`
Structure	circular DLL with a header (sentinel) node	non-circular DLL with `first`, `last`	hash table plus a DLL through the entries	DLL of 64-slot blocks	hash table plus a circular DLL with a sentinel
Both ends	$O(1)$	$O(1)$	oldest and newest entry, $O(1)$	$O(1)$	oldest and newest entry, $O(1)$
Insert or delete at a position	$O(1)$ via iterator	$O(1)$ via `ListIterator`	$O(1)$ by key	$O(n)$	$O(1)$ by key
Backward traversal	`rbegin()`, `rend()`	`descendingIterator()`, `reversed()`	`reversed()` (Java 21)	`reversed()`	`reversed()`
Move a node without copying	`splice`	no	automatic in access order	`rotate` (whole deque)	`move_to_end`

***Table 9.4**: GNU `std::list` is exactly the sentinel list of Section 9.7; Java's `LinkedHashMap` in access order and Python's `OrderedDict` and `lru_cache` ship the LRU pattern ready-made. Book: Section 9.12.*

Profitina in Practice

Browsing a history both ways: the analytics dashboard steps week by week with `cur = cur->prev` or `cur->next`, $O(1)$ per click.

Keeping the most recent items: the feed and notification list are a `collections.deque` with `maxlen`.

Caching expensive results: an LRU cache (`functools.lru_cache`, `OrderedDict`) is the sentinel list plus hash map of **Figure 9.17**.

Profitina is an AI visibility (GEO) platform whose server is a Python application built on FastAPI. Book: Section 9.13.

Profitina Laundry: One Week Back

Listing 9.6 From `main` in `laundry_weeks.cpp`: a cursor that moves one week back or forward.

```
    Week *cur = h.latest;  // the dashboard opens on the newest week
    std::printf("open:\n");
    report(cur);
    // b = one week back, f = one week forward
    const char *clicks = "bbbfbbbbb";
    for (const char *c = clicks; *c != '\0'; ++c) {
        // one step either way: O(1)
        Week *to = (*c == 'b') ? cur->prev : cur->next;
        if (to == nullptr) {
            std::printf("%s: no more weeks, stays on %s\n",
                        *c == 'b' ? "back" : "forward", cur->label.c_str());
            continue;
        }
        cur = to;
        std::printf("%s:\n", *c == 'b' ? "back" : "forward");
        report(cur);
    }
```

***Listing 9.6**: every click and every comparison with the previous week is one pointer step; at the oldest week `prev` is null and Back does nothing. Book: Section 9.13.*

Common Mistakes

Updating one direction only, or **overwriting the link you navigate by** in `insertAfter`.

Forgetting to clear `prev` at the new front (or `next` at the new back), and **forgetting the second external pointer**.

Testing for null in a circular list: `while (p != nullptr)` never ends.

Deleting, leaking or copying the sentinel; **believing $O(1)$ deletion means $O(1)$ deletion by value**.

Using a doubly linked list where an array would do: small elements and sequential reading favor the array.

Each mistake in the book comes with the figure or section that shows it. Book: Section 9.14.

Summary: Five Things to Remember

1. Two pointers per node; every forward link has a backward twin (Equation (9.1)), and every operation must keep it.

2. With `head` and `tail`, every end operation is $O(1)$, and so are inserting and deleting at a held node; insertion changes four pointers, deletion two.

3. A sentinel in a circular list removes every special case: no null pointer, no moving external pointer.

4. Deque and LRU list are sentinel lists; the LRU list adds a hash map from key to node.

5. The price is 8 bytes per node and cache locality; print both ways, check the invariant, run the tools.

Book: Section 9.15.

Next Time

Chapter 10 — Recursion. A function that calls itself on a smaller problem: base case, recursive case, and the call stack that makes it work.

Linked lists are recursive structures: a list is empty, or a node followed by a list.

Before then: work Exercises 9.1, 9.3, 9.8 and 9.10, in C++ and then in Java or Python.

The solutions, with programs, are in the book. Book: Sections 9.16 and 9.17.

Lecture 10 — Recursion

Course: Data Structure (EF234201)

Book: Irfan Subakti, *Data Structure with C++, Java and Python*, Chapter 10

Lecturer: Irfan Subakti — yifana@gmail.com

Today recursion stops being something that appeared here and there and becomes a tool we can trust, prove and measure. Book: Chapter 10, Sections 10.1–10.16.

What This Meeting Covers

1. **Thinking recursively** — base case, recursive case, induction, and the call stack frame by frame.
2. **Five classics** — factorial, Fibonacci (and six ways to tame it), Euclid's gcd, fast exponentiation, the Towers of Hanoi.
3. **Recursion on arrays and linked lists** — peel one element, or cut in half.
4. **Recursion trees** — the cost of a recursion, level by level, and a table of recurrences.
5. **The practical questions** — tail calls, depth limits in C++, Java and Python, and replacing recursion by an explicit stack.

Each classic adds one idea: the simplest recursion, a recursion that explodes, one that shrinks very fast, halving instead of subtracting, and one that is optimal yet exponential. Book: Section 10.1.

Where We Left Off

Recursion has been waiting in the wings: binary search (Section 2.5.5), merge sort and quicksort (Sections 3.8–3.9).

Chapter 5 solved the Towers of Hanoi with three stacks and a loop, and showed the **call stack** with three frames of `countdown`.

Chapter 7 computed the length of a list and reversed it recursively, warning that each node costs a stack frame.

Chapter 9 ended with `functools.lru_cache`, whose most famous use we meet today.

Chapter 11, on trees, will use today's ideas on almost every page: a tree is a root with smaller trees below it. Book: Section 10.1.

A Question Passed Along a Queue

In the Saturday queue at the Pustaka Ceria lending desk, you ask the person ahead: “How many people are ahead of *you*?” and add one.

Everybody runs the same procedure: if nobody is ahead, answer 0; otherwise ask and answer one more.

Each question is smaller: the person asked has fewer people ahead.

Someone answers without asking: the person at the desk says “zero” at once.

While the question travels, everyone who asked stands waiting, remembering to add one. In a program those waiting people are stack frames. Book: Section 10.2.1.

The Question Goes Forward, the Answers Come Back

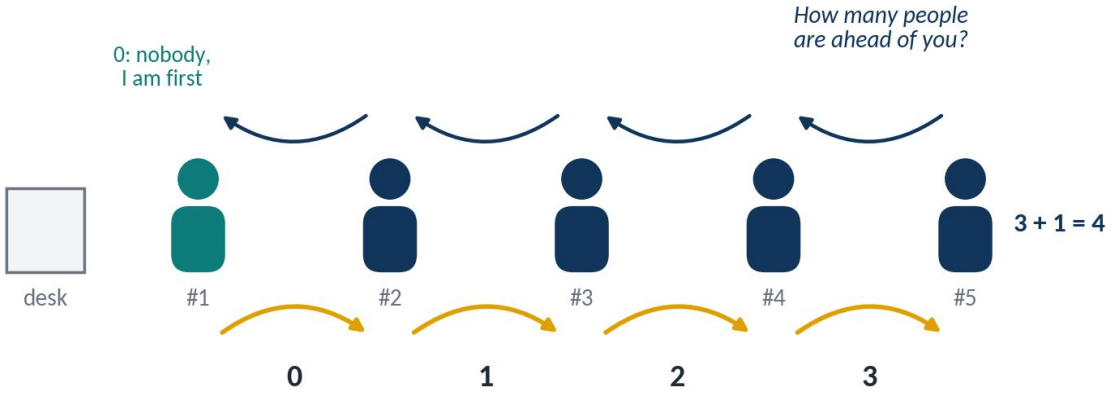


Figure 10.1 Counting the people ahead by asking: each question goes one place forward (navy), the first person answers 0 without asking anyone, and each answer comes back increased by one (amber).

Figure 10.1: five people waiting at the deepest moment, then the answers 0, 1, 2, 3, 4 travel back. Book: Section 10.2.1 and Figure 10.1.

Recursive Definitions

$$f(0)=3,\, f(n)=2f(n-1)+3 \text{ for } n\geq 1. \tag{10.1}$$

> **DEFINITION.** A *recursive definition* defines an object in terms of smaller instances of itself. Its *base case* gives the answer for the smallest instances directly; its *recursive case* gives the answer for a larger instance in terms of one or more **strictly smaller** instances. A *recursive function* is a function whose body calls the function itself.

Unfolding Equation (10.1): $f(0)=3$, $f(1)=9$, $f(2)=21$, $f(3)=45$, $f(4)=93$.

"Strictly" matters: $g(n)=g(n)+1$ *defines nothing, and* $h(n)=h(n+1)-1$ *never reaches a base case. Book: Section 10.2.2 and Equation (10.1).*

Proving It Right: Induction

The values 3, 9, 21, 45, 93 suggest the closed form Equation (10.2):

$$f(n) = 6 \cdot 2^n - 3 \text{ for every } n \geq 0. \tag{10.2}$$

Base case. $n = 0$: $6 \cdot 2^0 - 3 = 3 = f(0)$.

Inductive step. If $f(n-1) = 6 \cdot 2^{n-1} - 3$, then $f(n) = 2(6 \cdot 2^{n-1} - 3) + 3 = 6 \cdot 2^n - 3$.

> **KEY INSIGHT.** Do not trace every call: check the base case, *assume* the recursive calls are right for smaller inputs, and check the combination. That is an induction proof.

Recursion computes by reducing to smaller cases; induction proves by building up from them. Programmers call this the recursive leap of faith; induction justifies it. Book: Section 10.2.3 and Equation (10.2).

From Definition to Program

Listing 10.1 The function `f` of `01_recursive_f.cpp`: Equation (10.1) as C++, and the trace printed by its twin `fTraced`.

```cpp
long long f(int n) {
    if (n == 0) return 3;     // base case
    return 2 * f(n - 1) + 3;  // recursive case: a smaller n
}
```

```
call f(3)
    call f(2)
        call f(1)
            call f(0)
            f(0) returns 3
        f(1) returns 9
    f(2) returns 21
f(3) returns 45
```

Listing 10.1: four calls are unfinished at once; they finish in the opposite order to the one in which they started. The program then checks `f` against the closed form for $n=0$ to 10. Book: Section 10.2.4.

Three Rules, and What Breaks

1. A base case exists and is tested first. 2. Every call makes progress toward it, in a measure that cannot decrease forever. **3. Trust the call, combine correctly.**

Break rule 2, `return n * factRec(n + 1);`, and the frames pile up until the stack is full:

C++: segmentation fault (Windows: stack overflow exception), no message. **Java**: `StackOverflowError`. **Python**: `RecursionError` after 1,000 levels.

> **WARNING.** A recursion that does not progress is not a slow program; it is a crash within a fraction of a second. Test 0, 1 and 2 first.

The price of recursion: one frame per waiting call, a few nanoseconds of bookkeeping per call, and, if written naively, the same subproblem solved again and again. Book: Sections 10.2.5 and 10.3.4.

Factorial: One Frame per Call

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ n \cdot (n-1)! & \text{if } n \geq 1. \end{cases} \tag{10.3}$$

When `fact(3)` calls `fact(2)`, the program pushes a new *frame*: the same function, a different *activation*, with its own copy of `n`.

Unfolding Equation (10.3): $1! = 1$, $2! = 2$, $3! = 6$, $4! = 4 \cdot 6 = 24$.

Nothing about the call stack is special to recursion, and that is the point: the frames keep the waiting calls apart. Book: Sections 10.3.1–10.3.2 and Equation (10.3).

Winding Up and Unwinding

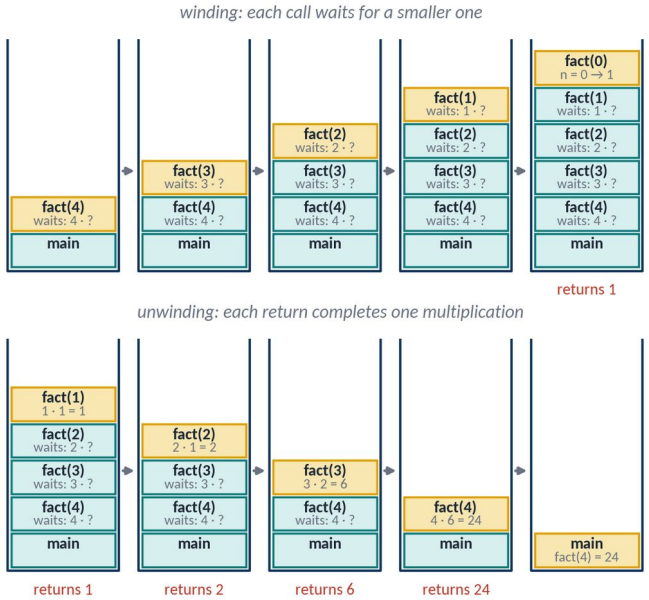


Figure 10.2 The call stack while `fact(4)` runs. Top row: every call waits for a smaller one until the base case `fact(0)` is on top. Bottom row: every return pops a frame and completes one multiplication in the frame below.

Figure 10.2: no multiplication on the way down; all four on the way back, in reverse order of the calls; the stack reaches $n+2$ frames. Book: Section 10.3.2 and Figure 10.2.

Factorial with a Loop and with Recursion

Listing 10.2 `factIter` and `factRec` from `02_factorial.cpp`.

```cpp
unsigned long long factIter(int n) {
    unsigned long long result = 1;
    for (int i = 2; i <= n; ++i) result *= i;  // 1 * 2 * 3 * ... * n
    return result;
}

unsigned long long factRec(int n) {
    if (n == 0) return 1;        // base case: 0! = 1
    return n * factRec(n - 1);  // recursive case: n! = n * (n-1)!
}
```

Listing 10.2: the loop multiplies into a running product; the recursion follows Equation (10.3) word for word.

Both agree up to $20!$*, the largest factorial in 64 bits;* $21!$ *silently wraps around to a plausible, wrong number. Java needs* `BigInteger`*; Python's integers never overflow. Book: Sections 10.3.3 and 10.4.1.*

What Each Costs

$$T(0)=c_0, T(n)=T(n-1)+c(n\geq 1). \tag{10.4}$$

Unfolding Equation (10.4): $T(n)=T(n-1)+c=\cdots=c_0+cn=\Theta(n)$. The loop is $\Theta(n)$ too.

Memory differs: the loop needs $O(1)$; the recursion keeps $n+1$ frames, $\Theta(n)$.

Pseudocode 10.1 Factorial with a loop and with recursion.

```
FACTORIAL-ITERATIVE(n)
 1.  result ← 1
 2.  for i ← 2 to n
 3.      result ← result · i
 4.  return result

FACTORIAL-RECURSIVE(n)
 1.  if n = 0
 2.      return 1                        // base case: 0! = 1
 3.  return n · FACTORIAL-RECURSIVE(n − 1)
```

Pseudocode 10.1: the time of a recursion is the work of all its calls; its extra memory is the largest number of frames on the stack at the same moment. A loop invariant is induction on the number of iterations. Book: Sections 10.4.1–10.4.2 and Equation (10.4).

Fibonacci's Rabbits

Liber Abaci, 1202: a newborn pair grows up in one month, then produces a new pair every month; no rabbit dies.

Pairs in month n = pairs in month $n-1$ (nobody dies) + pairs in month $n-2$ (the adults, each producing one), which is Equation (10.5):

$$F(0)=0, F(1)=1, F(n)=F(n-1)+F(n-2)(n\geq 2), \tag{10.5}$$

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144: 144 pairs in month 12, Fibonacci's answer.

The sequence was named by Édouard Lucas, the inventor of the Towers of Hanoi; it reappears in Euclid's algorithm and in the cost of its own naive computation. Book: Section 10.5.1 and Equation (10.5).

The Rabbits, Month by Month

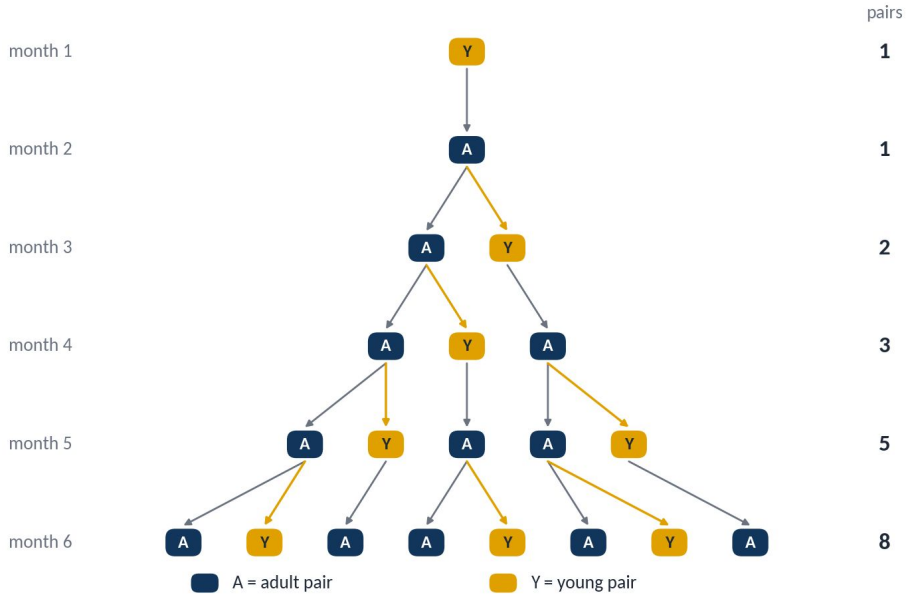


Figure 10.3 The first six months of Fibonacci's rabbits. Grey arrows show a pair living on into the next month; amber arrows show an adult pair producing a new young pair.

Figure 10.3: grey arrows, a pair living on; amber arrows, an adult pair producing a young pair. Book: Section 10.5.1 and Figure 10.3.

Naive Recursion Explodes

`fibNaive(n)` returns `fibNaive(n - 1) + fibNaive(n - 2)`: correct, and one of the worst algorithms in this book. Its calls obey Equation (10.6):

$$C(0)=C(1)=1, C(n)=C(n-1)+C(n-2)+1(n\geq 2). \tag{10.6}$$

$$C(n)=2F(n+1)-1. \tag{10.7}$$

$$F(n)\geq \varphi^{n-2} \text{ for every } n\geq 1. \tag{10.8}$$

By Equation (10.7) and Equation (10.8), the calls grow as $\Theta(\varphi^n)$ with $\varphi\approx 1.618$: every $+5$ in n multiplies the work by about 11. Both are two-line inductions. Book: Section 10.5.2 and Equations (10.6)–(10.8).

The Call Trees of $F(5)$

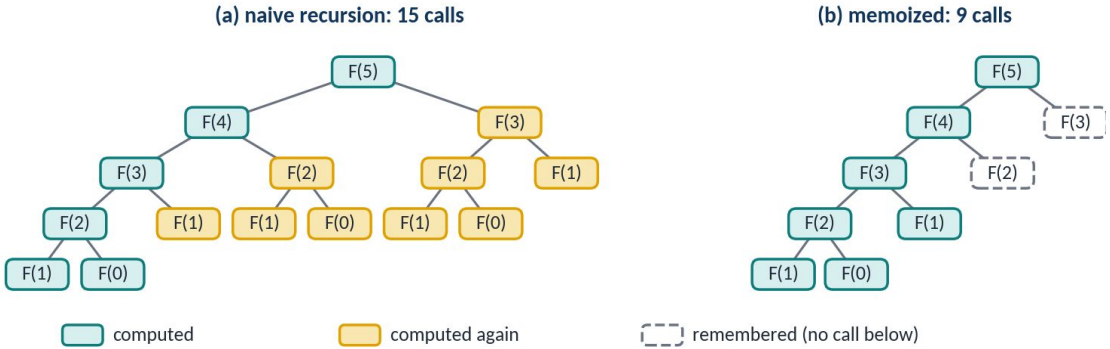


Figure 10.4 The calls made to compute $F(5)$. (a) Naive recursion: 15 calls, most of them recomputing values already known. (b) With memoization: 9 calls; a value already in the table is returned without any call below it.

Figure 10.4: (a) 15 calls, $F(3)$ twice, $F(2)$ three times; (b) with memoization, 9 calls. Book: Sections 10.5.2–10.5.3 and Figure 10.4.

Memoization: Remember What You Computed

Pseudocode 10.2 Fibonacci numbers by memoized recursion.

```
FIB-MEMO(n, memo)                       // memo[0..n] initialised to "unknown"
 1.  if n ≤ 1
 2.      return n
 3.  if memo[n] is known
 4.      return memo[n]                  // remembered: no recursion below
 5.  memo[n] ← FIB-MEMO(n − 1, memo) + FIB-MEMO(n − 2, memo)
 6.  return memo[n]
```

Pseudocode 10.2 computes each $F(k)$ once: $2n-1$ calls, $\Theta(n)$ time and memory. This is *top-down dynamic programming*.

> **KEY INSIGHT.** An exponential recursion is often exponential only because it solves the same subproblems again. If the *distinct* subproblems are few, store each answer the first time.

In Python, `@functools.lru_cache(maxsize=None)` memoizes a function automatically, in the sentinel list of Section 9.12.3. Book: Section 10.5.3.

Bottom-Up, and the Tail-Recursive Twin

Listing 10.3 From `03_fibonacci.cpp`: two variables, and the same loop written as a tail recursion.

```
ll fibTwoVars(int n) {  // bottom-up, only two values kept
    ll a = 0, b = 1;    // a = F(i), b = F(i+1)
    for (int i = 0; i < n; ++i) {
        ll next = a + b;
        a = b;
        b = next;
    }
    return a;
}

ll tailCalls = 0;
ll fibAux(ll x, ll y, int n) {  // x = F(i), y = F(i+1), n steps left
    ++tailCalls;
    if (n == 0) return x;
    return fibAux(y, x + y, n - 1);  // tail call: nothing left to do
}
ll fibTail(int n) { return fibAux(0, 1, n); }
```

Listing 10.3: each step replaces (a, b) by $(b, a+b)$. In `fibAux` the recursive call is the very last action, a tail call; `fibNaive` and `factRec` are not tail recursive. Book: Sections 10.5.4–10.5.5.

Fast Doubling and Binet's Formula

$$F(2k)=F(k)(2F(k+1)-F(k)),F(2k+1)=F(k)^2+F(k+1)^2. \tag{10.9}$$

Equation (10.9) gives $F(n)$ from $F(\lfloor n/2 \rfloor)$: $\lfloor \lg n \rfloor + 2$ calls, 8 for $n=90$.

$$F(n)=\frac{\varphi^n-\psi^n}{\sqrt{5}}. \tag{10.10}$$

TRAP. Equation (10.10) in `double` is exact only up to $F(70)$: at $n=71$ it answers 308,061,521,170,130 instead of 308,061,521,170,129.

Check the identities with $k=3$: $F(6)=2(2\cdot3-2)=8$, $F(7)=2^2+3^2=13$. Exact integer answers need exact integer arithmetic. Book: Sections 10.5.6–10.5.8 and Equations (10.9), (10.10).

Seven Ways to Compute $F(n)$

Table 10.1 Seven ways to compute $F(n)$, counting each operation on machine words as one step.

Method	Calls or steps	Time	Extra memory	Exact?
Naive recursion	$2F(n+1)-1$ calls	$\Theta(\varphi^n)$	$\Theta(n)$ stack	yes
Memoized recursion	$2n-1$ calls	$\Theta(n)$	$\Theta(n)$ table and stack	yes
Bottom-up table	$n-1$ additions	$\Theta(n)$	$\Theta(n)$ table	yes
Two variables	n additions	$\Theta(n)$	$O(1)$	yes
Tail recursion	$n+1$ calls	$\Theta(n)$	$\Theta(n)$ stack, $O(1)$ if the compiler eliminates the tail call	yes
Fast doubling	$\lfloor \lg n \rfloor + 2$ calls	$\Theta(\log n)$	$\Theta(\log n)$ stack	yes
Binet, `double`	one `pow`	$O(1)$	$O(1)$	only up to $F(70)$

Table 10.1: all exact methods agree on $F(90)=2,880,067,194,370,816,120$; $F(92)$ is the largest that fits in a signed 64-bit integer. Book: Section 10.5.9.

Measured: Calls and Time

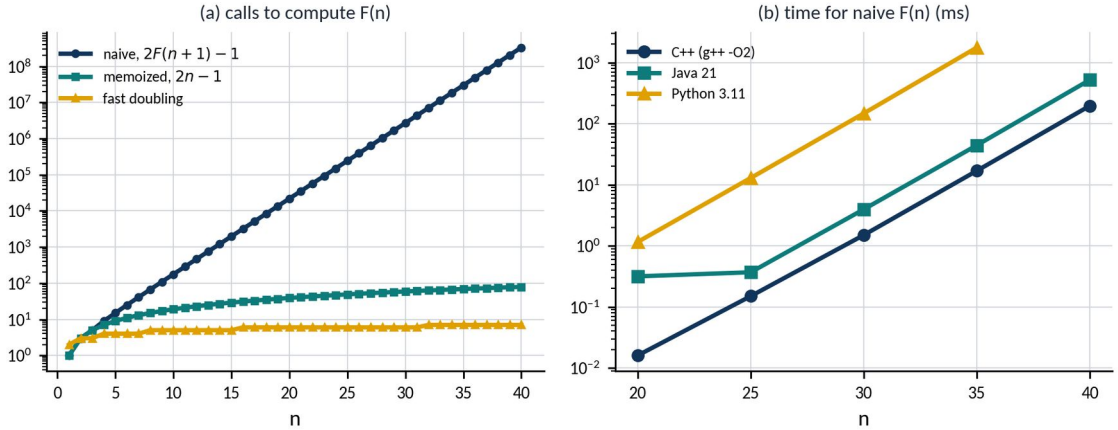


Figure 10.5 (a) Calls made to compute $F(n)$ by the three recursive methods, logarithmic scale. (b) Measured time of the naive recursion in C++, Java and Python, September 2026, logarithmic scale.

Figure 10.5: exponential growth is a straight line on a log scale; each language rises by about $\varphi^5 \approx 11$ every five steps. The slope belongs to the algorithm. Book: Section 10.5.9 and Figure 10.5.

Euclid's Greatest Common Divisor

$$\gcd(a,0)=a,\gcd(a,b)=\gcd(b,a \bmod b)(b>0), \tag{10.11}$$

Why true. With $a=qb+r$: a divisor of a and b divides $r=a-qb$; a divisor of b and r divides $a=qb+r$. Same common divisors, same greatest one.

Why it stops. The second argument goes from b to $a \bmod b<b$ and is never negative.

Equation (10.11) is from Euclid's Elements, about 300 BC; Knuth calls it "the granddaddy of all algorithms". Book: Section 10.6.1 and Equation (10.11).

Worked Example: $\gcd(228, 90)$

$$228 = 2 \cdot 90 + 48 \Rightarrow \gcd(228, 90) = \gcd(90, 48)$$

$$90 = 1 \cdot 48 + 42 \Rightarrow \gcd(90, 48) = \gcd(48, 42)$$

$$48 = 1 \cdot 42 + 6 \Rightarrow \gcd(48, 42) = \gcd(42, 6)$$

$$42 = 7 \cdot 6 + 0 \Rightarrow \gcd(42, 6) = \gcd(6, 0) = 6$$

The answer is the last non-zero remainder, 6, after four divisions. With the arguments swapped, $\gcd(90, 228)$*, the first division just swaps them back. Book: Section 10.6.2.*

Euclid as Cutting Squares

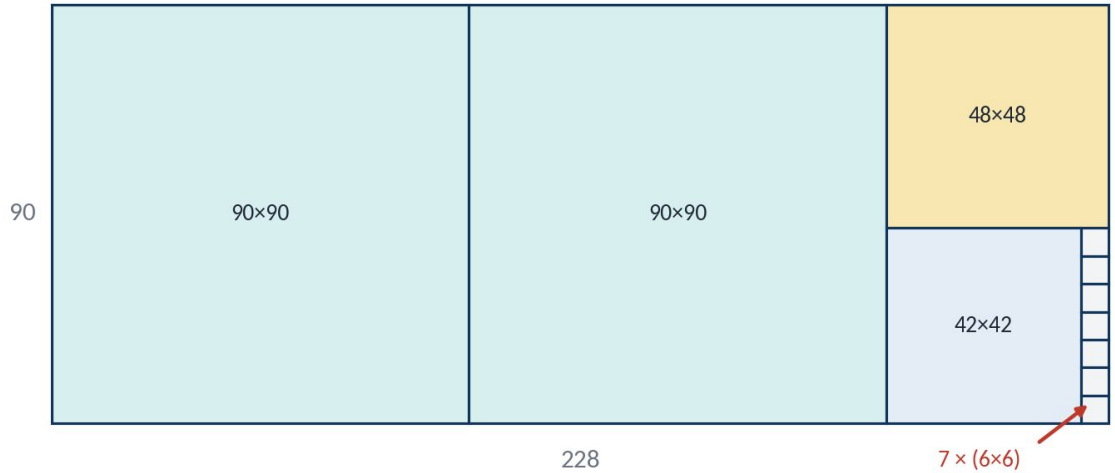


Figure 10.6 Euclid's algorithm on 228 and 90 as cutting squares from a rectangle: each division removes as many squares of the shorter side as fit; the last square, 6×6, tiles what remains exactly, so $\gcd(228, 90) = 6$.

Figure 10.6: the smallest square, 6×6, measures both sides a whole number of times ($228 = 38 \cdot 6$, $90 = 15 \cdot 6$). Book: Section 10.6.2 and Figure 10.6.

Recursive and Iterative, and Lamé's Theorem

Pseudocode 10.3 Euclid's algorithm, recursive and iterative.

```
GCD-RECURSIVE(a, b)
 1.  if b = 0
 2.      return a
 3.  return GCD-RECURSIVE(b, a mod b)     // a tail call

GCD-ITERATIVE(a, b)
 1.  while b ≠ 0
 2.      (a, b) ← (b, a mod b)
 3.  return a
```

THEOREM (Lamé, 1844). If $a > b \geq 1$ and Euclid's algorithm performs k divisions, then $a \geq F(k+2)$ and $b \geq F(k+1)$.

$$k \leq 1 + \log_{\varphi} b \approx 1 + 4.785 \log_{10} b\,. \tag{10.12}$$

Pseudocode 10.3: the recursive call is a tail call, so the loop is easy. By Equation (10.12) the divisions are at most about five times the decimal digits of b: at most 93 for 64-bit numbers. The worst case below 100 is $\gcd(89, 55)$, 9 divisions. Book: Sections 10.6.3–10.6.4.

Fast Exponentiation: Halve, Don't Subtract

$$m^0 = 1, m^n = m \cdot m^{n-1} (n \geq 1) \tag{10.13}$$

$$m^n = \begin{cases} 1 & \text{if } n = 0, \\ (m^{\lfloor n/2 \rfloor})^2 & \text{if } n \text{ is even}, \\ (m^{\lfloor n/2 \rfloor})^2 \cdot m & \text{if } n \text{ is odd}. \end{cases} \tag{10.14}$$

Equation (10.13) needs n multiplications; Equation (10.14) makes **one** call on half the exponent and squares the result.

Calling it twice, `power(m, n/2) * power(m, n/2)`, would give back all the work saved. Book: Sections 10.7.1–10.7.2 and Equations (10.13), (10.14).

Worked Example: 3^{13}

Pseudocode 10.4 Exponentiation by repeated squaring.

```
FAST-POWER(m, n)
 1.  if n = 0
 2.      return 1
 3.  half ← FAST-POWER(m, ⌊n/2⌋)          // one call on half the exponent
 4.  if n is even
 5.      return half · half
 6.  return half · half · m
```

Down: $13 \to 6 \to 3 \to 1 \to 0$. Up: $3^1 = 1 \cdot 1 \cdot 3 = 3$, $3^3 = 3 \cdot 3 \cdot 3 = 27$, $3^6 = 27 \cdot 27 = 729$, $3^{13} = 729 \cdot 729 \cdot 3 = 1{,}594{,}323$.

Pseudocode 10.4: five calls instead of fourteen. In binary $13 = 1101_2$: square for every digit, multiply by m for every 1. Book: Section 10.7.2.

Subtracting One versus Halving

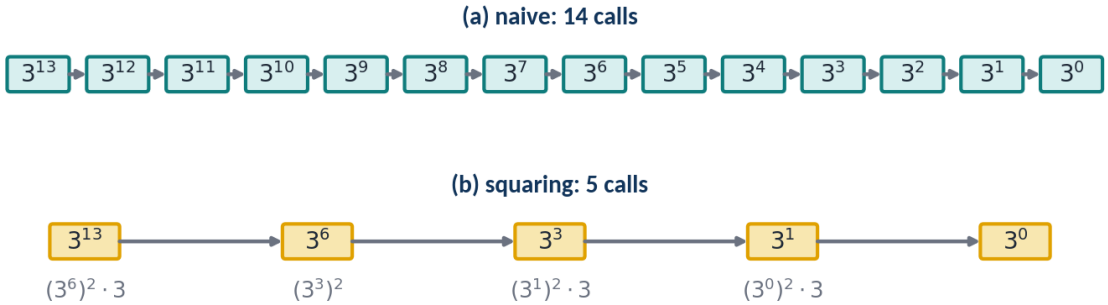


Figure 10.7 Computing 3^{13}. (a) Subtracting one from the exponent: 14 calls. (b) Halving the exponent: 5 calls; below each call, how its result is built from the next.

Figure 10.7: below each call, how its result is built from the next. Book: Section 10.7.2 and Figure 10.7.

Logarithmic Calls, and Powers Modulo p

$$\text{calls}(n) = \lfloor \lg n \rfloor + 2 \,(n \geq 1), \quad \text{multiplications} \leq 2(\lfloor \lg n \rfloor + 1). \tag{10.15}$$

Equation (10.15): 5 calls for $n = 13$, 31 for 10^9, 61 for 10^{18}.

Reduce after every multiplication, since $(x\,y) \bmod p = ((x \bmod p)(y \bmod p)) \bmod p$; with $p = 1{,}000{,}000{,}007$ every product fits in 64 bits.

Listing 10.4 `powerFast` from `06_power.cpp`.

```
ull powerFast(ull m, int n) {  // m^n = (m^(n/2))^2, times m if n is odd
    ++calls;
    if (n == 0) return 1;
    ull half = powerFast(m, n / 2);
    if (n % 2 == 0) return half * half;
    return half * half * m;
}
```

Listing 10.4: $7^{10^{18}} \bmod p = 259{,}616{,}729$ *in 61 calls. RSA in every TLS connection computes such powers; Python's* `pow(7, n, p)` *does the same. Book: Sections 10.7.2–10.7.3.*

The Towers of Hanoi, Recursively

To move n disks from `from` to `to` with helper `via`:

1. move the $n-1$ smaller disks from `from` to `via`; **2.** move disk n from `from` to `to`; **3.** move the $n-1$ disks from `via` to `to`.

Steps 1 and 3 are the very problem we are solving, but *smaller*: that is all recursion needs. The largest disk never blocks a move.

Section 5.2 noticed it already: solving three disks is solving two disks twice, with one move in between. The base case is $n=0$, nothing to move. Book: Section 10.8.1.

The Recursive Idea, Drawn

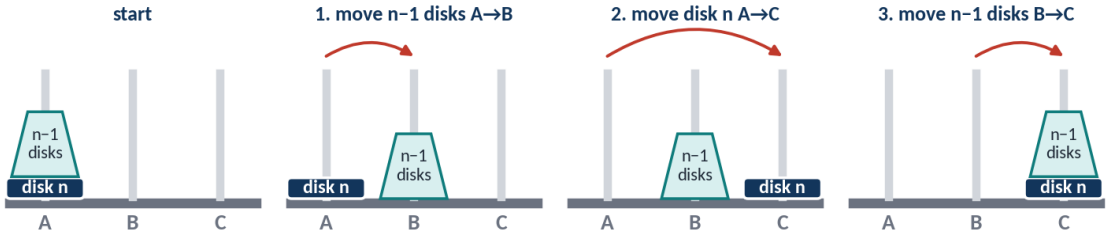


Figure 10.8 The recursive solution of the Towers of Hanoi: move the $n-1$ smaller disks out of the way, move the largest disk, and move the $n-1$ disks back on top of it.

Figure 10.8: the $n-1$ smaller disks drawn as one block. Book: Section 10.8.1 and Figure 10.8.

Hanoi in Pseudocode and in C++

Pseudocode 10.5 The recursive Towers of Hanoi.

```
HANOI(n, from, to, via)
 1.  if n = 0
 2.      return                          // base case: nothing to move
 3.  HANOI(n − 1, from, via, to)         // clear the n − 1 smaller disks onto via
 4.  move disk n from `from` to `to`
 5.  HANOI(n − 1, via, to, from)         // bring them back on top of disk n
```

Listing 10.5 `hanoi` from `07_hanoi.cpp`; each move is played on three `std::stack<int>` pegs that reject illegal moves.

```cpp
void hanoi(int n, char from, char to, char via) {
    if (n == 0) return;  // base case: nothing to move
    ++depth;
    if (depth > maxDepth) maxDepth = depth;
    hanoi(n - 1, from, via, to);  // 1. clear the n-1 smaller disks onto via
    moveDisk(n, from, to);        // 2. move the largest disk
    hanoi(n - 1, via, to, from);  // 3. bring the n-1 disks back on top of it
    --depth;
}
```

Pseudocode 10.5 and Listing 10.5: no `ILLEGAL MOVE` in 2,097,130 moves for 1 to 20 disks, and the depth is only n. Java uses `ArrayDeque<Integer>`, Python a list. Book: Section 10.8.2.

Worked Example: The Order of the Moves

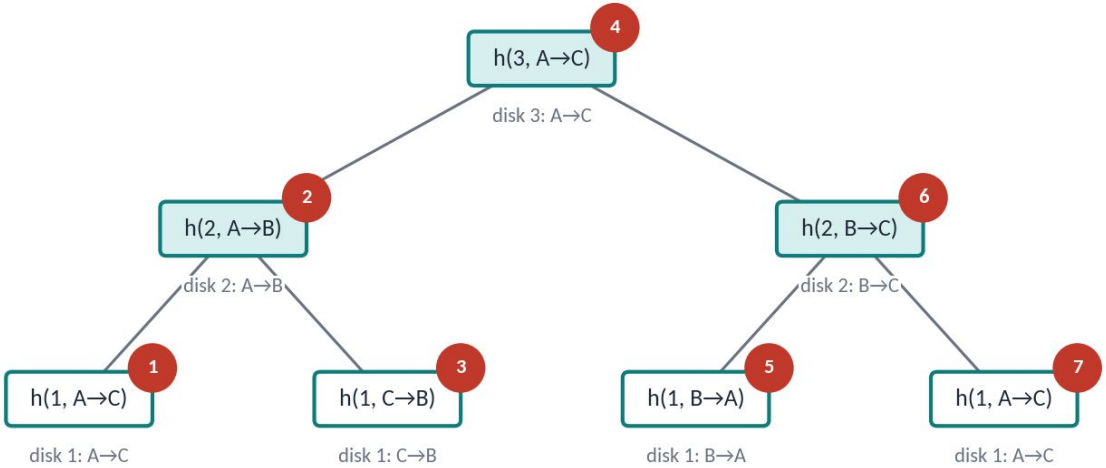


Figure 10.9 The recursion tree of `hanoi(3, A, C, B)`; `h(k, X→Y)` moves k disks from X to Y. Each call makes its own move between the moves of its two children; the red numbers give the order of the moves.

Figure 10.9: left subtree, own move, right subtree (an in-order traversal, Chapter 11): disk 1 A→C, 2 A→B, 1 C→B, 3 A→C, 1 B→A, 2 B→C, 1 A→C. Book: Section 10.8.3 and Figure 10.9.

Exactly $2^n - 1$ Moves, and No Fewer

$$M(0) = 0, M(n) = 2M(n-1) + 1 (n \geq 1). \tag{10.16}$$

Upper bound: $M(n) = 2(2^{n-1} - 1) + 1 = 2^n - 1$ by induction.

Lower bound: before the *first* move of disk n, all $n-1$ smaller disks sit on the third peg; after its *last* move they must all come back. So $L(n) \geq 2L(n-1) + 1 \geq 2^n - 1$.

The algorithm is **optimal**: its exponential cost belongs to the problem. Unlike naive Fibonacci, its two calls do not overlap, so there is nothing to remember.

Equation (10.16) for 64 disks gives 18,446,744,073,709,551,615 moves, about 585 billion years at one move per second. Book: Section 10.8.4 and Equation (10.16).

Recursion on Arrays

Peel off one element: $A[0..n-1]$ is $A[0..n-2]$ plus $A[n-1]$; base case the empty array; depth n.

Cut in half: $A[lo..hi]$ is two halves around $mid = lo + \lfloor (hi - lo)/2 \rfloor$; depth $\lceil \lg n \rceil + 1$.

`sumArr` peels; `maxArr` halves; `binSearch` halves and discards; `reverseArr` swaps the ends and recurses inside; `isPalindrome` checks the end letters and the word between them.

On the loan counts 12, 7, 31, 4, 25, 18 of Pustaka Ceria: sum 97, max 31; "racecar", "rotor", "a" and "" are palindromes, "library" is not. Book: Section 10.9.1.

A Linked List Is a Recursive Structure

Listing 10.6 From `08_arrays_lists.cpp`: printing backward, and removing every node that holds a key.

```
void printBackward(const Node *p) {  // recurse, then print
    if (p == nullptr) return;
    printBackward(p->next);
    std::printf(" %d", p->data);
}

Node *removeAll(Node *p, int key) {  // returns the new first node
    if (p == nullptr) return nullptr;
    Node *rest = removeAll(p->next, key);
    if (p->data == key) {
        delete p;
        return rest;
    }
    p->next = rest;
    return p;
}
```

Listing 10.6: recurse first, print afterward, and the stack plays the role of the missing `prev` links. `removeAll` returns the new head, so `head = removeAll(head, 40);` needs no special case. Valgrind: all heap blocks freed. Book: Sections 10.9.2–10.9.3.

Recursion Trees

One node per call, its recursive calls as children, and in each node the work the call does *itself*.

Total time = sum of all nodes, computed level by level; stack memory = height of the tree.

$$T(n) = 2T(n/2) + cn \Rightarrow T(n) = cn(\lg n + 1) = \Theta(n \lg n). \tag{10.17}$$

Merge sort, Equation (10.17): $\lg n + 1$ levels of cost cn. Hanoi: level i has 2^i moves, total $2^n - 1$.

Halving gives few levels; shrinking by one gives n levels, with the nodes doubling on each. Book: Sections 10.10.1–10.10.2 and Equation (10.17).

Two Trees, Two Totals

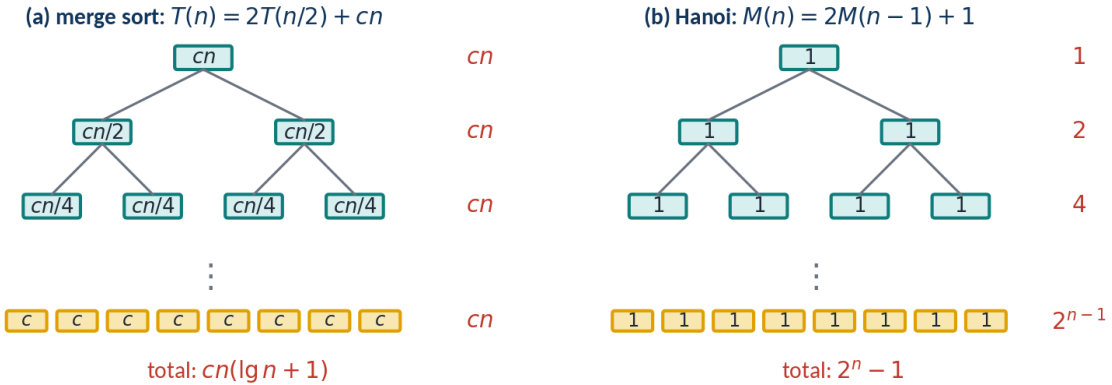


Figure 10.10 Recursion trees with the cost of each level on the right. (a) Merge sort: $\lg n + 1$ levels of cost $c\,n$ each. (b) Hanoi: level i costs 2^i, and the levels add up to $2^n - 1$.

Figure 10.10: the cost of each level on the right. Book: Section 10.10.2 and Figure 10.10.

A Table of Recurrences

Table 10.2 Common recurrences, their solutions, and the algorithms of this book that produce them.

Recurrence	Solution	Where it arises
$T(n)=T(n-1)+c$	$\Theta(n)$	factorial, list length, linear search, m^n naive
$T(n)=T(n/2)+c$	$\Theta(\log n)$	binary search, fast power, fast doubling
$T(n)=2T(n/2)+c$	$\Theta(n)$	maximum by halving (`maxArr`)
$T(n)=2T(n/2)+cn$	$\Theta(n\log n)$	merge sort (Equation (10.17))
$T(n)=T(n-1)+cn$	$\Theta(n^2)$	selection sort written recursively, quicksort's worst case
$T(n)=2T(n-1)+c$	$\Theta(2^n)$	Towers of Hanoi (Equation (10.16))
$T(n)=T(n-1)+T(n-$	$\Theta(\varphi^n)$	naive Fibonacci (Equation (10.6))

Table 10.2: `maxArr` halves but is still $\Theta(n)$, because it must look at every element; halving saves time only when a half can be discarded. Book: Section 10.10.3.

Tail Calls

```
long long sumPlain(long long n) {                // 1 + 2 + ... + n
    if (n == 0) return 0;
    return n + sumPlain(n - 1);                  // NOT a tail call: the + waits
}

long long sumAcc(long long n, long long acc) {  // acc + 1 + 2 + ... + n
    if (n == 0) return acc;
    return sumAcc(n - 1, acc + n);               // a tail call
}
```

`sumAcc` carries the partial sum in an *accumulator*; its frame is never needed again, so the call can become a jump: *tail-call elimination*.

GCC at `-O2`: `sumAcc` becomes a loop, and so does `sumPlain` (GCC introduces the accumulator itself). At `-O0` both crash at ten million.

The option is `-foptimize-sibling-calls`, on at `-O2`, `-O3`, `-Os`. The C++ standard promises nothing: never rely on it. Book: Sections 10.11.1–10.11.3.

One Frame Instead of Six

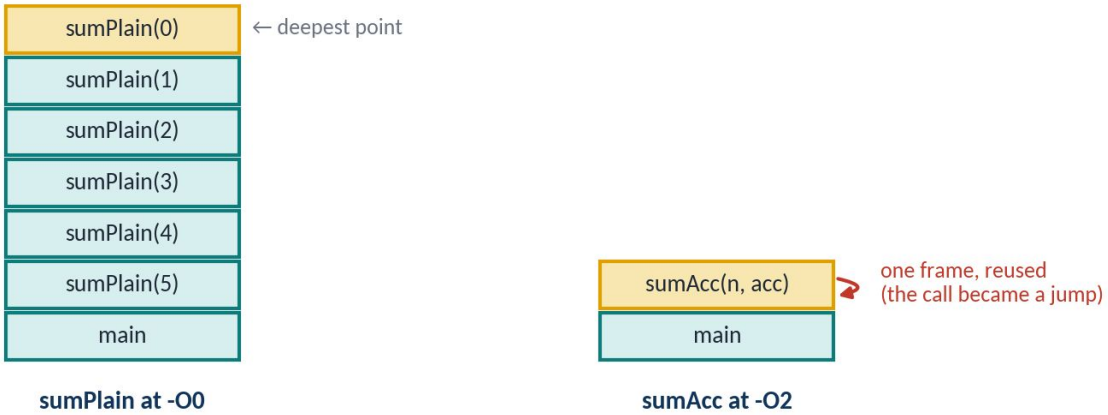


Figure 10.11 Left: without optimization, `sumPlain(5)` has six frames of its own on the stack at the deepest point. Right: at `-O2` the tail call of `sumAcc` is compiled as a jump, and a single frame serves the whole recursion (GCC does the same for `sumPlain` after introducing an accumulator).

Figure 10.11: after tail-call elimination one frame serves the whole recursion. Book: Section 10.11.2 and Figure 10.11.

The JVM and CPython Do Not Eliminate Tail Calls

```
recursion limit: 1000
         n        sum_plain          sum_acc         sum_loop
       900           405450           405450           405450
      1000   RecursionError   RecursionError           500500
     10000   RecursionError   RecursionError         50005000
   1000000   RecursionError   RecursionError     500000500000
```

In Java both `sumPlain` and `sumAcc` throw `StackOverflowError` at $n = 10^6$; `sumLoop` succeeds.

Python's creator declined tail-call elimination because it destroys the stack traces programmers debug with.

The rule: in Java and Python a tail-recursive function is as deep as its input; if you want a loop, write the loop. Book: Section 10.11.4.

How Deep Can Recursion Go?

$$d_{\max} \approx \frac{S}{s} \tag{10.18}$$

C++: 8 MB stack on Linux and macOS, 1 MB on Windows; frames of 72 bytes at `-O2`: 116,000 levels succeed, 117,000 crash.

Java: `-Xss`, default 1,024 KB; measured 10,730 to 59,020 levels, varying with the JIT.

Python: a counted limit, 1,000 by default (`RecursionError` at depth 997); `sys.setrecursionlimit` and `threading.stack_size` raise it.

Equation (10.18) predicted $8,388,608/72 \approx 116,508$ *levels. The depth limit of a C++ program even depends on its optimization level. Book: Section 10.12 and Equation (10.18).*

Recursion or Iteration?

Table 10.3 Recursion or iteration: a decision for each algorithm of this chapter.

Algorithm	Depth	Recommendation
factorial, m^n naive, sum of $1..n$	n	loop (tail recursion by accumulator, then a loop)
Fibonacci	n (and exponential time if naive)	two variables, or fast doubling for huge n
Euclid's gcd, fast power, binary search	$O(\log n)$	recursion is fine; the loop is equally easy
merge sort, maxima by halving	$O(\log n)$	recursion
Towers of Hanoi	n, with n tiny	recursion
list length, search, copy, print backward	n	loop, or an explicit stack when the order matters
nested menus, trees (Chapter 11), directories	the nesting depth	recursion if the depth is known to be small; an explicit stack if the input comes from outside

Table 10.3: convert when the depth is proportional to the input and the input may be large; keep the recursion when the depth is logarithmic or small and known. Book: Section 10.13.1.

An Explicit Stack of Frames

Listing 10.7 The loop of `hanoiIter` in `11_explicit_stack.cpp`: our own stack of frames, each with a stage.

```
    while (!st.empty()) {
        Frame &f = st.back();
        if (f.n == 0) {  // base case: return at once
            st.pop_back();
        } else if (f.stage == 0) {  // first recursive call
            f.stage = 1;
            Frame sub = {f.n - 1, f.from, f.via, f.to, 0};
            st.push_back(sub);  // f is not used after this: push_back may move it
        } else {                // the first half has returned
            Frame done = f;
            st.pop_back();  // nothing is left to do after the second call,
            ++moves;        // so this frame can go before it is made
            if (print)
                printf("%2lld. Move disk %d from rod %c to rod %c\n", moves, done.n,
                       done.from, done.to);
            st.push_back({done.n - 1, done.via, done.to, done.from, 0});
        }
        if (st.size() > maxStack) maxStack = st.size();
    }
```

Listing 10.7: a `Frame` holds n, the pegs and a stage; the second call is a tail call, so the frame is popped before it is made. At most $n+1$ frames; the stack lives in the heap. Book: Sections 10.13.2–10.13.3.

The Explicit Stack, Pass by Pass

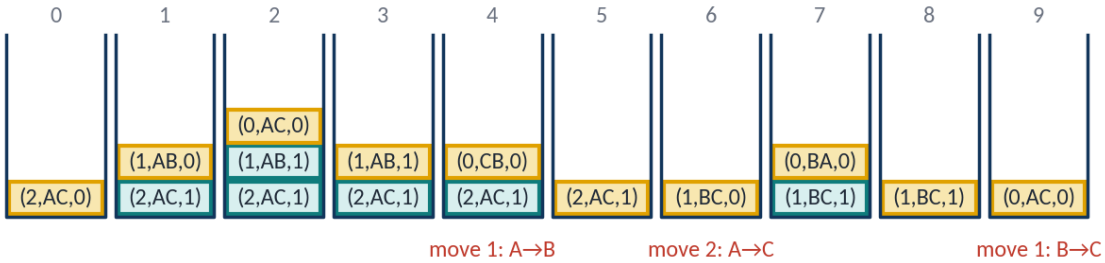


Figure 10.12 The explicit stack while `hanoiIter(2)` runs; each frame shows (n, source and target peg, stage), and the top frame is highlighted. Each column is the stack after one pass of the loop; the moves appear below the pass that makes them.

Figure 10.12: after pass 9 the stack is empty and all three moves are made. A million-node list is walked backward the same way, with a `vector` of pointers. Book: Sections 10.13.3–10.13.4 and Figure 10.12.

Profitina in Practice

Walking a site: “visit a page, then every page it links to” is recursive, but link chains can be thousands deep; a crawler keeps its own stack or queue, a seen-set and a depth limit.

Nested structured data: JSON-LD nests objects to any depth; checking it is a recursive walk, with a depth guard because the input comes from outside.

Remembering results: memoization with a bounded LRU cache pays off exactly when the same subproblem recurs.

Profitina is an AI visibility (GEO) platform built with FastAPI; we describe the shape of each problem, not the product’s code. Book: Section 10.14.

Profitina Laundry's Nested Menu

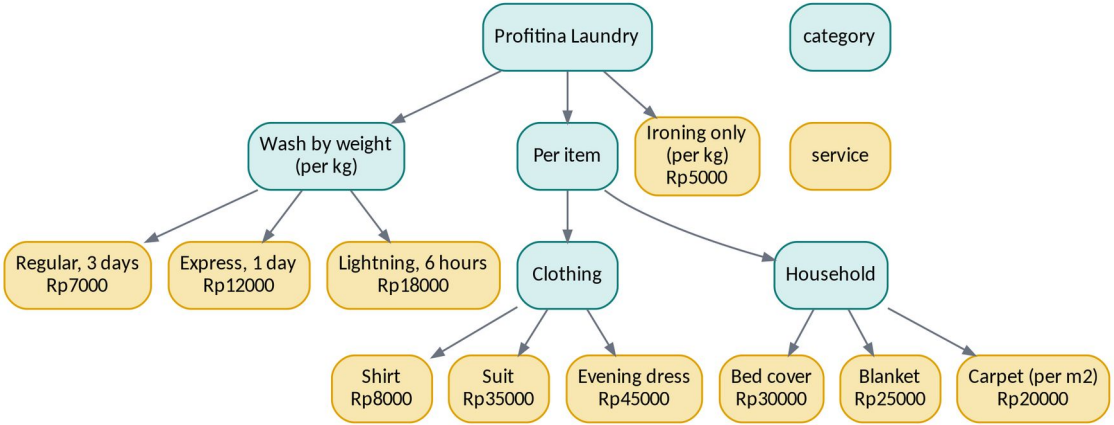


Figure 10.13 The service menu of Profitina Laundry: categories (teal) contain services (amber) and smaller categories, to any depth.

Figure 10.13: categories contain services and smaller categories, to any depth: a tree, the subject of Chapter 11. Book: Section 10.14 and Figure 10.13.

Counting the Services

Listing 10.8 `countServices` and `height` from `12_laundry_menu.cpp`.

```
int countServices(const Item &it) {
    if (it.children.empty()) return 1;  // base case: a service
    int total = 0;
    for (const Item &c : it.children) total += countServices(c);
    return total;
}

int height(const Item &it) {  // levels below the top
    int h = 0;
    for (const Item &c : it.children) h = max(h, 1 + height(c));
    return h;
}
```

```
services: 10 recursively, 10 with an explicit stack
levels below the top: 3
cheapest service: Ironing only (per kg), Rp5000
```

Listing 10.8: base case "a service, no children", recursive case "a category: combine the children". `countServicesIter` *gets the same 10 with an explicit stack. Book: Section 10.14.*

Common Mistakes

No base case, or one that is jumped over; **a call that does not shrink the problem**.

Recursing over long lists: one frame per node overflows every default stack.

Overlapping subproblems (memoize), and **calling twice what should be called once** (`power(m, n/2) * power(m, n/2)`).

Trusting floating point, **forgetting overflow**, **relying on tail-call elimination**, **raising Python's limit without thinking**.

Losing the returned value: call `head = removeAll(head, key)`, not `removeAll(head, key)`.

Test the smallest inputs first: 0, 1, 2 and any negative value the function might receive. Book: Section 10.15.

Summary: Five Things to Remember

1. A base case, a strictly smaller recursive case, and an induction proof that trusts the smaller calls.

2. Time is the work of all calls; memory is the deepest pile of frames.

3. Overlapping subproblems explode ($\Theta(\varphi^n)$ for Fibonacci); memoization or bottom-up fixes them. Hanoi's $2^n - 1$ is optimal.

4. Halving gives logarithmic depth (Euclid, fast power, binary search); peeling gives depth n.

5. Tail calls are eliminated only by optimizing C++ compilers, never by guarantee; deep recursions become loops with an explicit stack.

Book: Section 10.16.

Next Time

Chapter 11 — Trees and Binary Search Trees. A root with smaller trees below it: traversals, search, insertion and deletion, all written recursively.

The in-order traversal of the Hanoi tree returns, now as a tool.

Before then: work Exercises 10.3, 10.4, 10.9 and 10.10, in C++ and then in Java or Python.

The solutions, with programs, are in the book. Book: Sections 10.17 and 10.18.

Lecture 11 — Trees and Binary Search Trees

Course: Data Structure (EF234201)

Book: Irfan Subakti, *Data Structure with C++, Java and Python*, Chapter 11

Lecturer: Irfan Subakti — yifana@gmail.com

Today we give up the line: each element may point to several elements below it, and a search can throw away a whole subtree at every step. Book: Chapter 11, Sections 11.1–11.14.

What This Meeting Covers

1. Trees — definition, vocabulary, notations, and two representations in memory.

2. Binary trees — shapes, how many nodes fit, how low a tree can be.

3. Traversals — preorder, inorder, postorder, level order, without recursion, and expression trees.

4. The binary search tree — insert, search, minimum and maximum, successor, deletion in three cases, and costs.

5. Shape and balance — measured heights, rotations, AVL and red-black trees, general trees, and the libraries.

The running example is `insert(65, 5, 70, 3, 10)`, built, searched, counted, printed and deleted from. Book: Section 11.1.

Where We Left Off: the Line's Dilemma

Every structure so far has been a **line**: arrays, stacks and queues, singly and doubly linked lists.

A **sorted array** searches in $O(\lg n)$ but shifts half of itself on every insertion.

A **linked list** inserts in $O(1)$ at a held position but finds that position in $\Theta(n)$.

We want both: as easy to search as a sorted array, as easy to change as a list. The tree gets there by giving up the line.

The shape is naturally recursive, which is why Chapter 10 came first: a tree is a root with smaller trees hanging from it. Book: Section 11.1.

Four Trees from Everyday Life

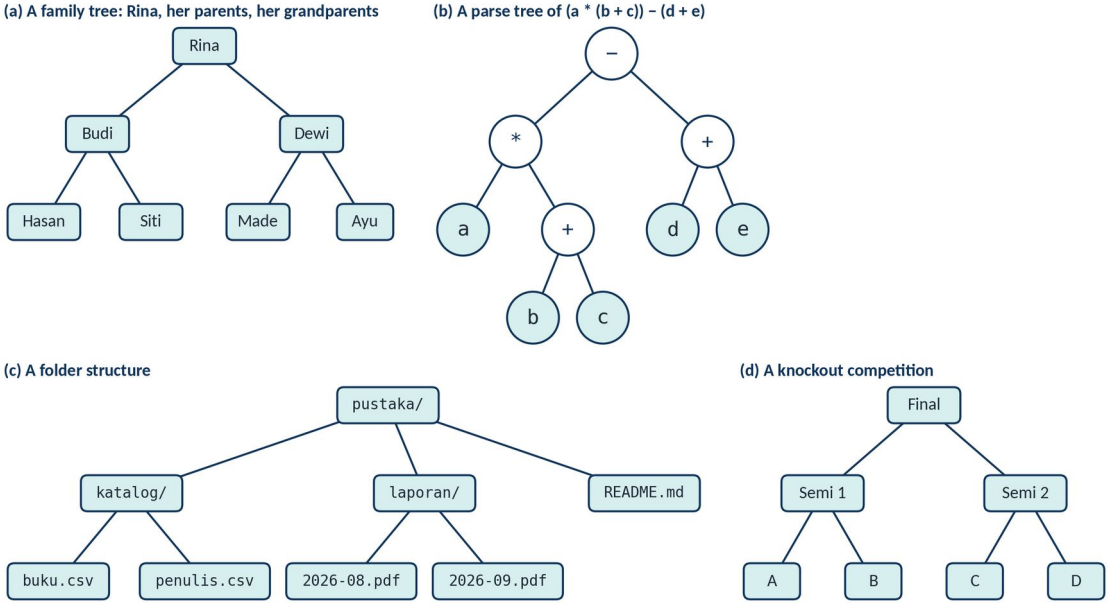


Figure 11.1 Four trees: (a) a family tree of ancestors, (b) the parse tree a compiler builds for an arithmetic expression, (c) the folders and files on a disk, (d) the draw of a knockout competition.

Figure 11.1: one element at the top, every other element below exactly one element, nothing in a circle. That is all a tree is. Book: Section 11.2.1 and Figure 11.1.

The Definition

> **DEFINITION.** A *tree* is a finite set T of one or more *nodes* such that (1) there is one specially designated node called the *root*, and (2) the remaining nodes are partitioned into $m \geq 0$ disjoint sets $T_1, \ldots, T_m$, each of which is itself a tree: the *subtrees* of the root.

The definition is **recursive**, and not circular: every subtree has fewer nodes. From the root to any node there is **exactly one path**.

A tree of n nodes has Equation (11.1) edges: every node except the root has exactly one edge up to its parent.

$$e = n - 1 \tag{11.1}$$

A quick check for every tree you draw: count the nodes and the lines; the lines must be one fewer. Zero or more disjoint trees form a forest. Book: Section 11.2.2 and Equation (11.1).

A Tree of Eleven Nodes

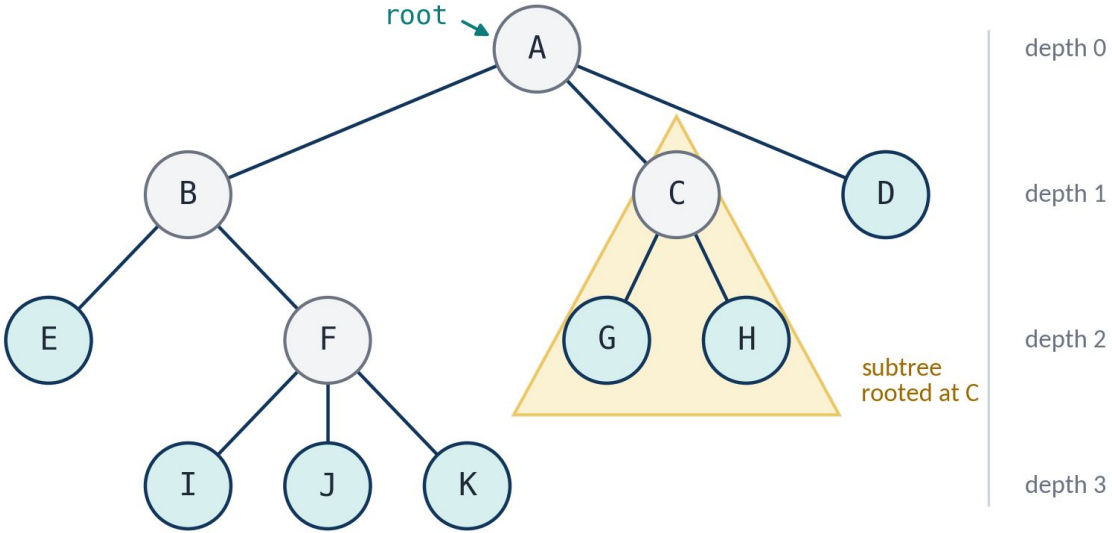


Figure 11.2 A tree of eleven nodes. Internal nodes are grey, leaves are teal; the amber triangle marks the subtree rooted at C, and the depth of each level is given on the right.

Figure 11.2 defines the vocabulary: root, parent, child, sibling, leaf, internal node, path, ancestor, descendant, subtree, depth, level, height, degree, size. Book: Section 11.2.3 and Figure 11.2.

The Vocabulary, Evaluated

Table 11.1 The vocabulary of trees, evaluated on the tree of Figure 11.2.

Term	Value in Figure 11.2
Root	A
Internal nodes	A, B, C, F
Leaves	E, I, J, K, G, H, D
Children of B	E, F
Siblings of G	H
Ancestors of J	F, B, A
Descendants of C	G, H
Depth of F; height of F	2; 1
Height of B; height of the tree	2; 3
Degree of A; degree of F; degree of the tree	3; 3; 3
Size; number of edges	11; 10, as Equation (11.1) requires

Table 11.1. Depth is measured down from the root; height up from the lowest leaf. F has depth 2 and height 1; B depth 1 and height 2. The empty tree has height -1. *Book: Sections 11.2.3–11.2.4.*

One Tree, Four Notations

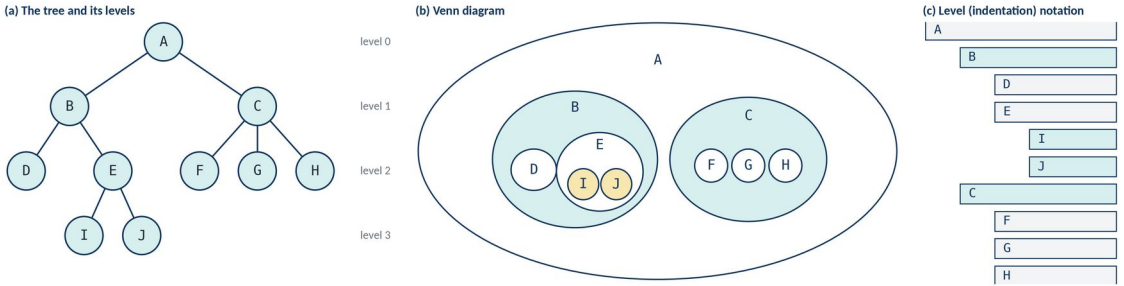


Figure 11.3 One tree in three notations: (a) the usual drawing with its levels, (b) a Venn diagram, in which every subtree is a set drawn inside its parent's set, (c) level notation, in which each node is a bar indented by its depth.

Figure 11.3; the fourth, parenthesis notation, writes the same tree as `(A(B(D,E(I,J)),C(F,G,H)))`*, the way LISP and JSON write trees. Book: Section 11.3 and Figure 11.3.*

The Static Representation: an Array

Number the positions level by level from 0 and store node i in slot i; children and parent come from Equation (11.2):

$$left(i) = 2i+1, right(i) = 2i+2, parent(i) = \lfloor \frac{i-1}{2} \rfloor \tag{11.2}$$

No pointers, and the parent in $O(1)$; but every missing position wastes a slot. A right chain of $h+1$ nodes needs $2^{h+1}-1$ slots.

A complete tree has no empty slots at all, which is why Chapter 14 stores its heaps this way. Book: Section 11.4.1 and Equation (11.2).

A Tree in an Array

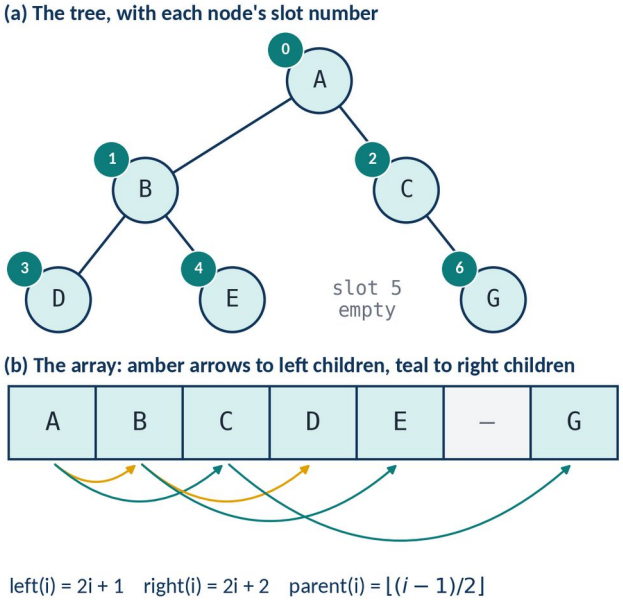


Figure 11.4 The static representation: (a) each node is numbered level by level, (b) node i is stored in slot i, and its children are found at slots $2i+1$ and $2i+2$ by Equation (11.2).

Figure 11.4: C has no left child, so slot $2 \cdot 2+1=5$ stays empty. `array_tree.cpp` *walks the tree in preorder and inorder by index arithmetic alone. Book: Section 11.4.1 and Figure 11.4.*

The Dynamic Representation: Linked Nodes

```
typedef struct MyNode {
    int data;
    struct MyNode *left;
    struct MyNode *right;
} Node;
```

The doubly linked node of Chapter 9 with new meanings: `left` and `right` point *down* to two subtrees; the tree is reached through `Node *root`.

Of the $2n$ pointer fields only $n-1$ hold an edge, so Equation (11.3) of them are null:

$$2n-(n-1)=n+1 \tag{11.3}$$

For any number of children: a parent array, a list of children (the DOM of a web page), or first child and next sibling (Section 11.10). Book: Sections 11.4.2–11.4.3 and Equation (11.3).

Linked Nodes, Drawn

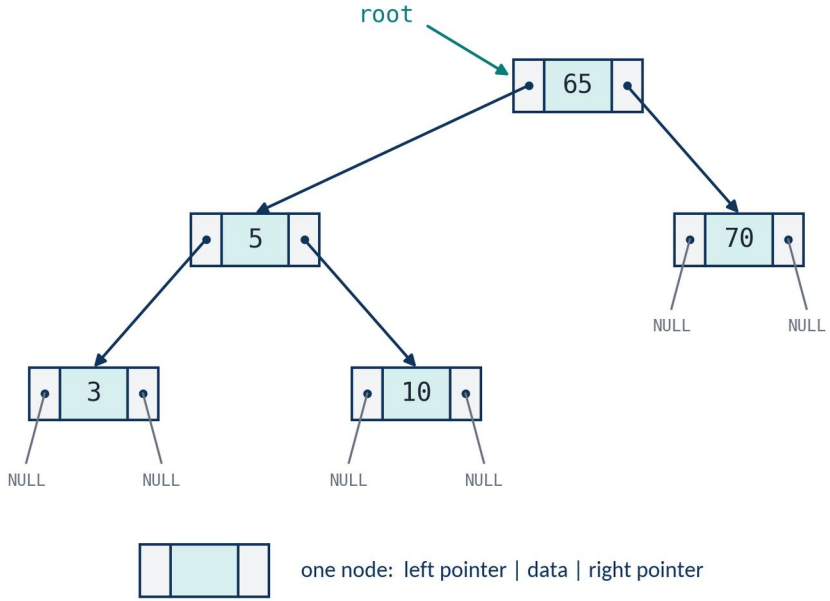


Figure 11.5 The dynamic representation of a binary tree: every node has a left pointer, a data field and a right pointer; `root` points at the top node, and every missing child is a NULL pointer.

Figure 11.5: five nodes, six NULL pointers, as the formula says. Book: Section 11.4.2 and Figure 11.5.

Binary Trees and Their Shapes

DEFINITION. A *binary tree* is either *empty*, or a root node together with two binary trees, its *left subtree* and its *right subtree*, which have no node in common.

DEFINITION. Full: every node has 0 or 2 children. **Perfect**: full, all leaves at the same depth. **Complete**: every level full except possibly the last, which is filled from the left. **Skewed**: every internal node has exactly one child.

A binary tree may be empty, and left and right are distinguished. Older texts use "full" for perfect and "complete" for full; this book follows Cormen et al. Book: Sections 11.5.1–11.5.2.

Parts of a Binary Tree

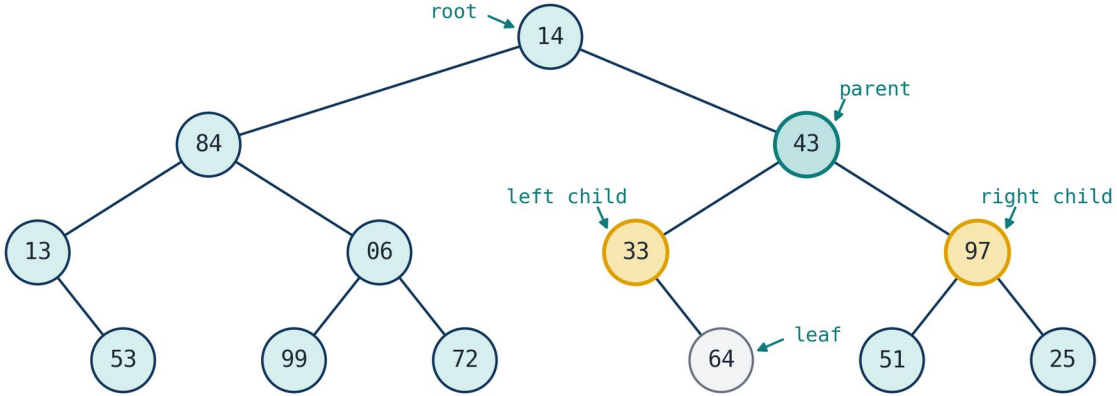


Figure 11.6 A binary tree of fourteen nodes: node 43 is the parent of its left child 33 and its right child 97, and 64, which has no children, is a leaf.

Figure 11.6: two children per node is exactly what a yes/no decision needs, and any tree can be stored as a binary tree. Book: Section 11.5.1 and Figure 11.6.

Six Shapes

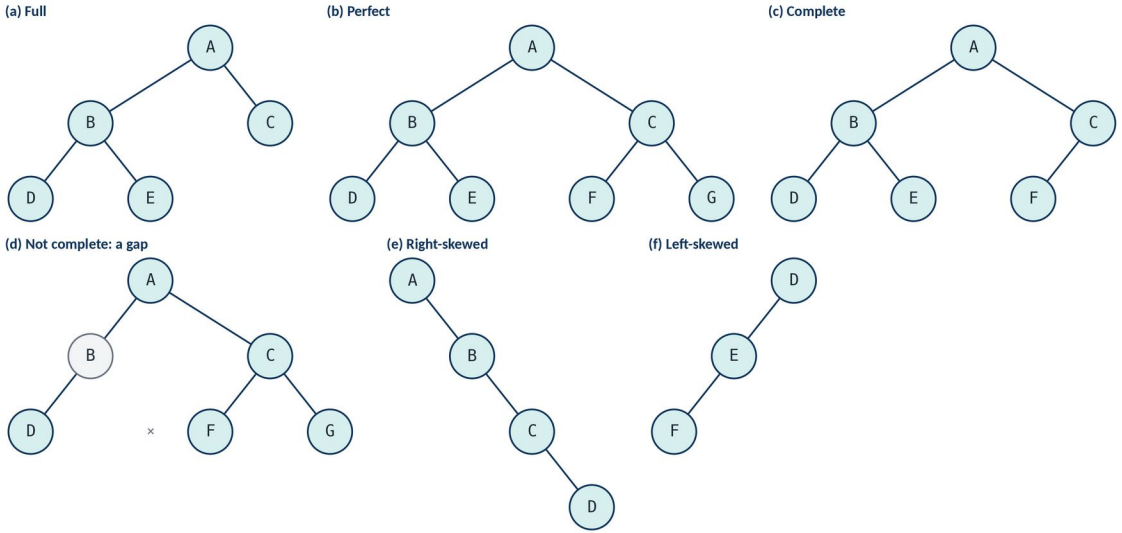


Figure 11.7 Shapes of binary trees: (a) full, (b) perfect, (c) complete, (d) not complete, because position 4 on the last level is empty while position 5 is used, (e) right-skewed, (f) left-skewed.

Figure 11.7: every perfect tree is full and complete; a skewed tree of n nodes is a chain of height $n - 1$, a linked list drawn diagonally. Book: Section 11.5.2 and Figure 11.7.

How Many Nodes Fit, How Low a Tree Can Be

$$\text{nodes on level } \ell \le 2^{\ell} \tag{11.4}$$

$$n \le 1+2+4+\cdots+2^{h} = \sum_{\ell=0}^{h} 2^{\ell} = 2^{h+1}-1 \tag{11.5}$$

$$h \ge \lceil \lg(n+1) \rceil - 1 = \lfloor \lg n \rfloor \tag{11.6}$$

$$L = I + 1 \tag{11.7}$$

Equation (11.4) and Equation (11.5) bound the nodes; turned around, Equation (11.6) says no binary tree of n nodes is lower than $\lfloor \lg n \rfloor$: 19 for a million nodes, against 999,999 for a chain. By Equation (11.7) a knockout of 64 teams needs 63 matches. Book: Section 11.5.3.

Three Depth-First Orders

Pseudocode 11.1 The three depth-first traversals; they differ only in the line on which the node itself is visited.

```
PREORDER(x):
 1.  if x ≠ NIL
 2.      visit x
 3.      PREORDER(x.left)
 4.      PREORDER(x.right)

INORDER(x):
 1.  if x ≠ NIL
 2.      INORDER(x.left)
 3.      visit x
 4.      INORDER(x.right)

POSTORDER(x):
 1.  if x ≠ NIL
 2.      POSTORDER(x.left)
 3.      POSTORDER(x.right)
 4.      visit x
```

Pseudocode 11.1: pre, the root before its subtrees; in, between them; post, after them. Base case the empty tree; each call gets a strictly smaller tree. Book: Section 11.6.1.

Worked Example: the Three Orders

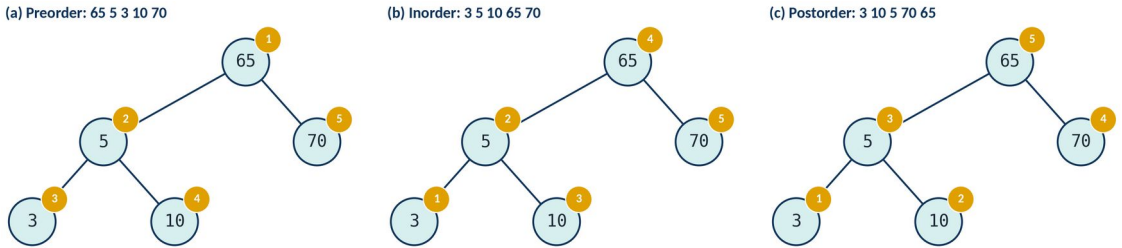


Figure 11.8 The order in which each traversal visits the nodes of the same tree; the amber badge on a node is its position in the visit sequence.

Figure 11.8: preorder 65, 5, 3, 10, 70; inorder 3, 5, 10, 65, 70 (sorted!); postorder 3, 10, 5, 70, 65. Preorder copies and saves, inorder lists sorted, postorder deletes and computes. Book: Section 11.6.1 and Figure 11.8.

A Walk Around the Tree

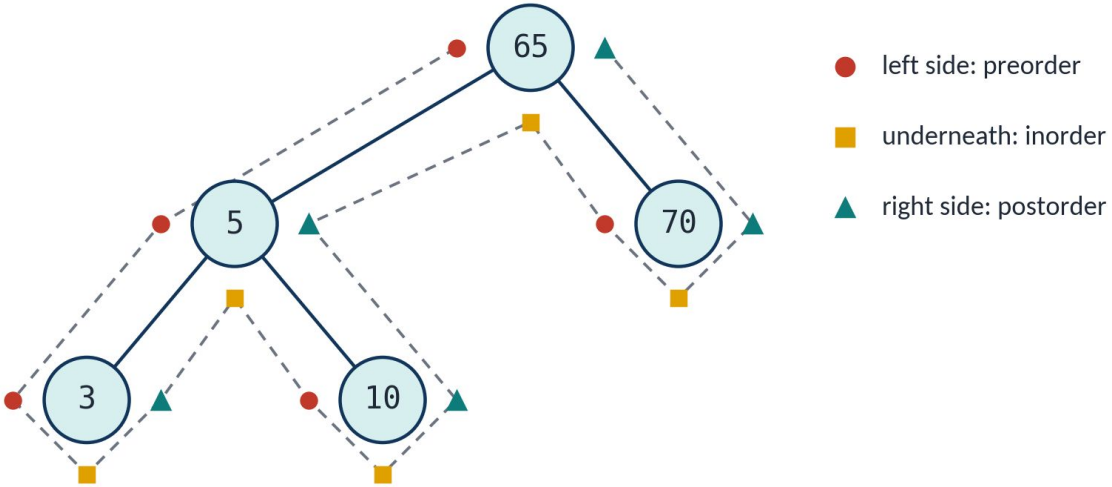


Figure 11.9 The walk around the tree touches each node three times: listing the nodes when they are passed on the left gives preorder, underneath gives inorder, and on the right gives postorder.

Figure 11.9: the Euler tour passes each node on its left (preorder), underneath (inorder) and on its right (postorder). The recursion is this walk. Book: Section 11.6.2 and Figure 11.9.

Level Order with a Queue

Pseudocode 11.2 `LEVEL-ORDER`: breadth-first traversal; children join the back of the queue, so a whole level is visited before any node of the next.

```
LEVEL-ORDER(T):
 1.  if T.root = NIL
 2.      return
 3.  Q ← an empty queue
 4.  ENQUEUE(Q, T.root)
 5.  while Q is not empty
 6.      x ← DEQUEUE(Q)              // the oldest waiting node
 7.      visit x
 8.      if x.left ≠ NIL
 9.          ENQUEUE(Q, x.left)
10.      if x.right ≠ NIL
11.          ENQUEUE(Q, x.right)
```

$$T(0)=c,T(n)=T(k)+T(n-k-1)+d(n\geq 1) \tag{11.8}$$

Pseudocode 11.2 is breadth-first: children join the back of the queue, so a whole level is visited before the next. By Equation (11.8) every traversal is $\Theta(n)$ whatever the shape; the space is $O(h)$ for depth-first and the widest level for level order. Book: Sections 11.6.3 and 11.6.5.

Level Order, Traced

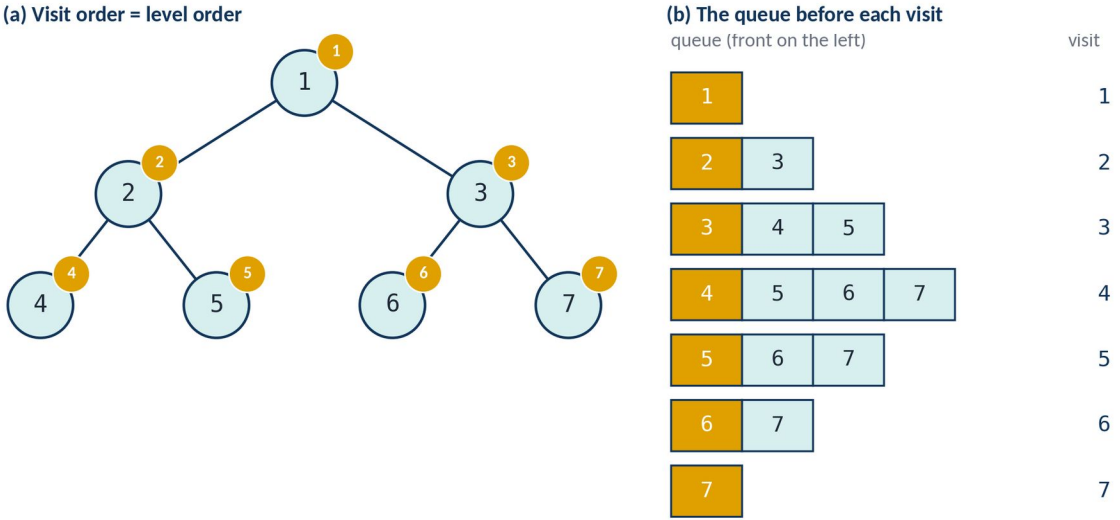


Figure 11.10 Level order on a perfect tree of seven nodes: (a) the visit order, (b) the queue just before each visit, with the node about to leave it in amber.

Figure 11.10: [1], [2, 3], [3, 4, 5], [4, 5, 6, 7], … The queue holds at most two consecutive levels; Chapter 13 generalizes this to breadth-first search. Book: Section 11.6.3 and Figure 11.10.

Inorder Without Recursion

Pseudocode 11.3 Inorder traversal with an explicit stack: go left pushing every node, pop and visit, then continue with the right subtree.

```
INORDER-ITERATIVE(T):
 1.  S ← an empty stack
 2.  cur ← T.root
 3.  while cur ≠ NIL or S is not empty
 4.      while cur ≠ NIL             // go down to the leftmost node
 5.          PUSH(S, cur)
 6.          cur ← cur.left
 7.      cur ← POP(S)                // its left subtree is finished
 8.      visit cur
 9.      cur ← cur.right             // now do its right subtree
```

Go left pushing every node; pop and visit; continue with the right subtree. Preorder: pop, visit, push right then left. Postorder: two stacks.

Pseudocode 11.3: an explicit stack on the heap cannot overflow the call stack on a degenerate tree, and an iterative traversal can pause, which is how `std::map` iterators work. Book: Section 11.6.4.

The Stack at Each Visit

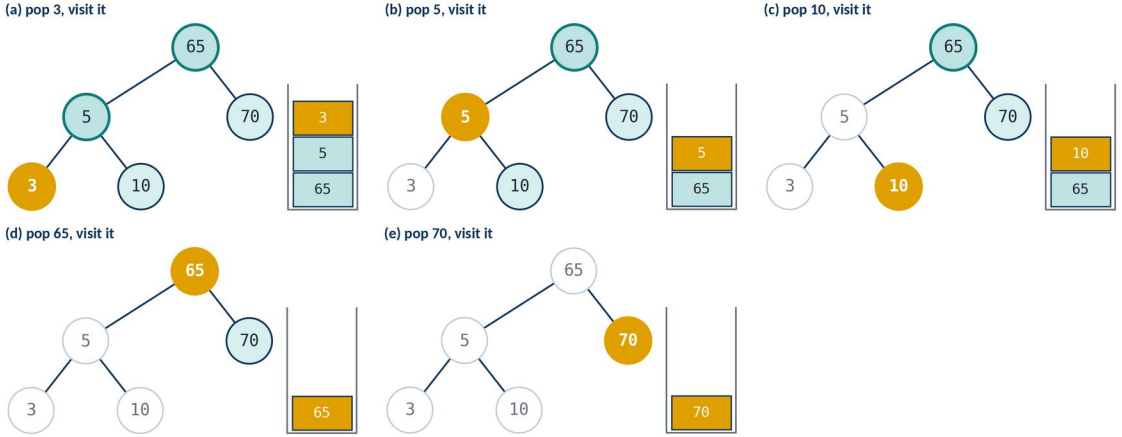


Figure 11.11 The explicit stack of Pseudocode 11.3 at each of the five visits: teal nodes are on the stack, the amber node is being visited, and grey nodes have already been visited.

Figure 11.11: push 65, 5, 3; visit 3; visit 5; push and visit 10; visit 65; push and visit 70. Book: Section 11.6.4 and Figure 11.11.

Expression Trees

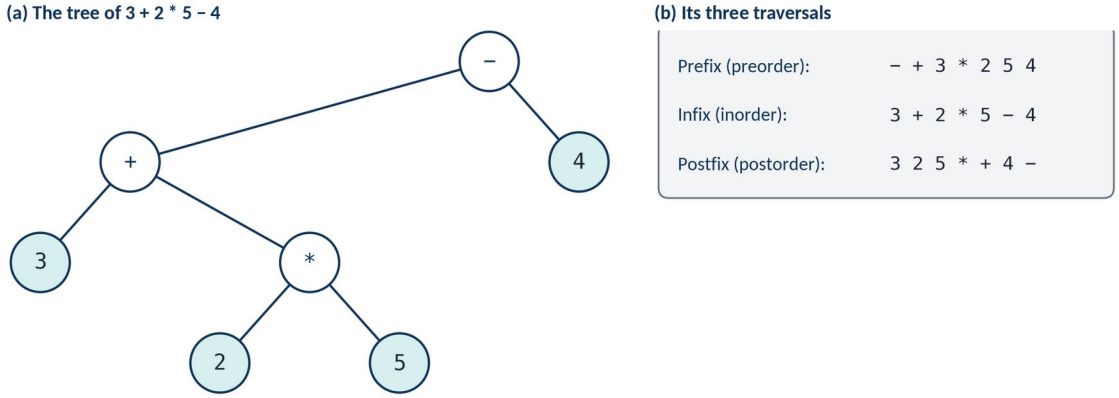


Figure 11.12 The expression tree of `3 + 2 * 5 - 4` (a) and its three depth-first traversals (b): preorder gives prefix notation, inorder gives infix notation, and postorder gives postfix notation.

Figure 11.12: preorder `- + 3 * 2 5 4` (prefix), inorder `3 + 2 * 5 - 4` (infix, needing parentheses), postorder `3 2 5 * + 4 -` (postfix). Evaluating is a postorder traversal. Book: Section 11.6.6 and Figure 11.12.

Building the Tree from Postfix

Pseudocode 11.4 Building an expression tree from postfix notation with a stack of trees.

```
BUILD-EXPRESSION-TREE(postfix):
 1.  S ← an empty stack of trees
 2.  for each symbol c of postfix, from left to right
 3.      z ← a new node holding c
 4.      if c is an operator
 5.          z.right ← POP(S)        // the right operand was pushed last
 6.          z.left ← POP(S)
 7.      PUSH(S, z)
 8.  return POP(S)                   // the only tree left
```

The stack algorithm of Section 5.9 with trees on the stack: an operand pushes a one-node tree; an operator pops the right, then the left operand and pushes the combined tree.

Pseudocode 11.4: `expr_tree.cpp` *evaluates* `3 2 5 * + 4 -` *to 9:* $2 \cdot 5 = 10$, $3 + 10 = 13$, $13 - 4 = 9$. *Compilers do this on a grand scale. Book: Section 11.6.6.*

The Stack of Trees

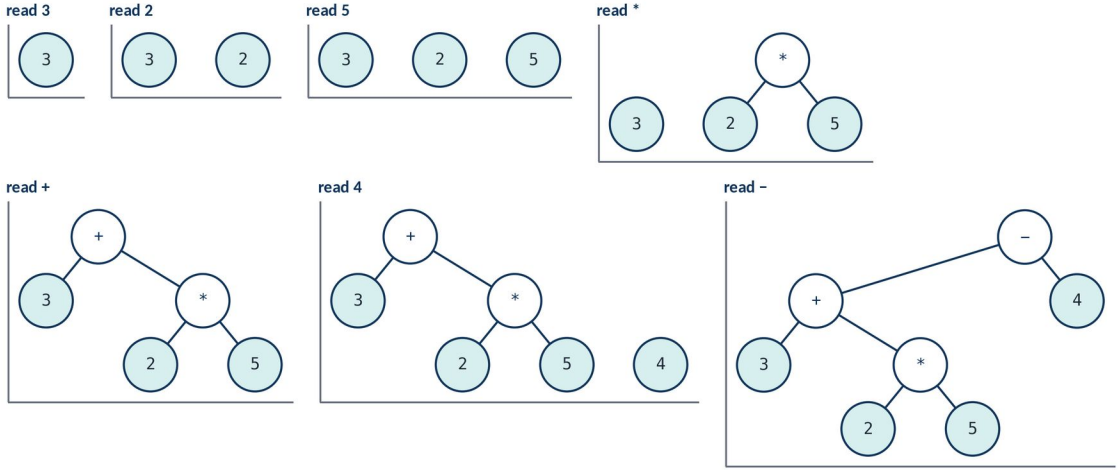


Figure 11.13 The stack of trees while `3 2 5 * + 4 -` is read: operands push one-node trees, and each operator combines the two newest trees into one.

Figure 11.13: at the end exactly one tree remains, the expression tree. Book: Section 11.6.6 and Figure 11.13.

The Binary Search Tree

DEFINITION. A *binary search tree* (BST) is a binary tree in which, for every node x, every key in the left subtree of x is less than $x.key$ and every key in the right subtree is greater.

Every key in the subtree, not only the child; and at *every* node, not only the root.

INORDER IS SORTED. By induction: left subtree ascending, then the root, then the right subtree ascending, and the BST property orders the three parts.

A BST is a dictionary that stays sorted: it lists keys in order, reports the minimum and maximum and finds neighbors, which a hash table cannot. Book: Section 11.7.1.

Binary Search, Taken Out of the Array

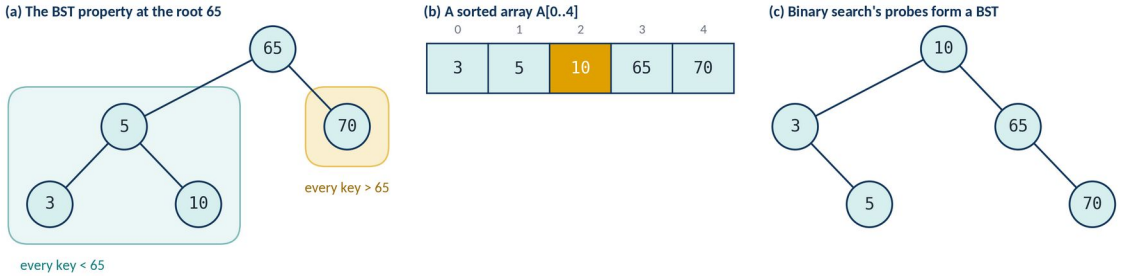


Figure 11.14 (a) The BST property at the root: every key of the left subtree is smaller than 65 and every key of the right subtree is larger; (b) and (c) the probes of binary search on a sorted array form a binary search tree.

Figure 11.14: draw every sequence of binary-search probes and you get a BST. Windley, Booth and Colin, and Hibbard found it independently around 1960. Book: Section 11.7.1 and Figure 11.14.

Insertion

Pseudocode 11.5 `TREE-INSERT`: walk down as in a search and hang the new node on the empty subtree where the walk ends.

```
TREE-INSERT(x, z):              // insert key z into the subtree whose root pointer is x
 1.  if x = NIL
 2.      x ← a new node with key z, left NIL, right NIL
 3.  else if z < x.key
 4.      TREE-INSERT(x.left, z)
 5.  else if z > x.key
 6.      TREE-INSERT(x.right, z)
 7.  else
 8.      report "z has existed in the tree"
```

Walk down as in a search; hang the new node on the empty subtree where the walk ends. Line 2 must change the *pointer* that was NIL.

Pseudocode 11.5: C uses `Node **root`*, C++ may use* `Node *&root`*, and Java and Python return the new subtree root:* `t.left = insert(t.left, z)`*. Cost* $O(h)$*. Book: Section 11.7.2.*

Worked Example: `insert(65, 5, 70, 3, 10)`

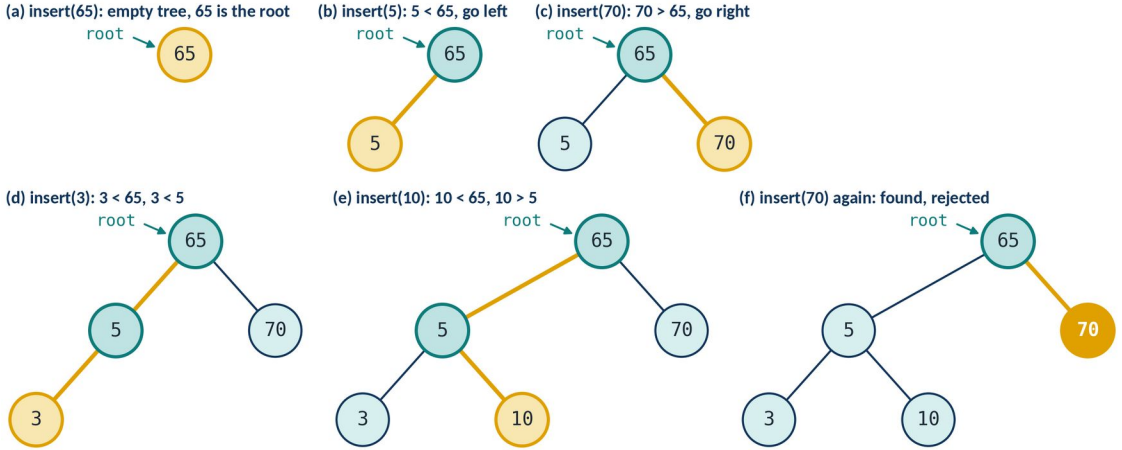


Figure 11.15 The tree after each call of `insert(65, 5, 70, 3, 10)` and a repeated `insert(70)`: teal nodes and amber edges are the path of the search, the amber node is the new leaf, and the duplicate 70 is found and rejected.

Figure 11.15: 65 is the root; 5 goes left; 70 right; 3 left, left; 10 left, right; a second 70 is found and rejected. The shape records the history of insertions. Book: Section 11.7.2 and Figure 11.15.

Insertion in C++

Listing 11.1 `insert` from `bst.cpp`: the address of a pointer, so the call can change the caller's pointer.

```
void insert(Node **root, int newData) {     // root is the ADDRESS of a pointer
    if (*root == nullptr) {                 // an empty subtree: the node goes here
        Node *newNode = new Node;
        newNode->data = newData;
        newNode->left = nullptr;
        newNode->right = nullptr;
        *root = newNode;                    // changes the caller's pointer
        std::printf("%d has been inserted by insert()\n", newData);
    } else if (newData < (*root)->data) {
        insert(&(*root)->left, newData);
    } else if (newData > (*root)->data) {
        insert(&(*root)->right, newData);
    } else {
        std::printf("%d has existed in the tree\n", newData);
    }
}
```

Listing 11.1: `&(*root)->left` *is the address of the current node's left field.* `bst.py` *and* `Bst.java` *return the new subtree root instead:* `t.left = insert(t.left, new_data)`*. Book: Sections 11.7.7–11.7.8.*

Search

Pseudocode 11.6 Searching a binary search tree, recursively and iteratively; both return the node holding k, or NIL.

```
TREE-SEARCH(x, k):
 1.  if x = NIL or k = x.key
 2.      return x
 3.  if k < x.key
 4.      return TREE-SEARCH(x.left, k)
 5.  else
 6.      return TREE-SEARCH(x.right, k)

ITERATIVE-TREE-SEARCH(x, k):
 1.  while x ≠ NIL and k ≠ x.key
 2.      if k < x.key
 3.          x ← x.left
 4.      else
 5.          x ← x.right
 6.  return x
```

Pseudocode 11.6: equal, found; smaller, go left; larger, go right; NIL, absent. The iterative form needs no stack, because it never comes back. Cost $O(h)$. Book: Section 11.7.3.

Worked Example: Two Searches

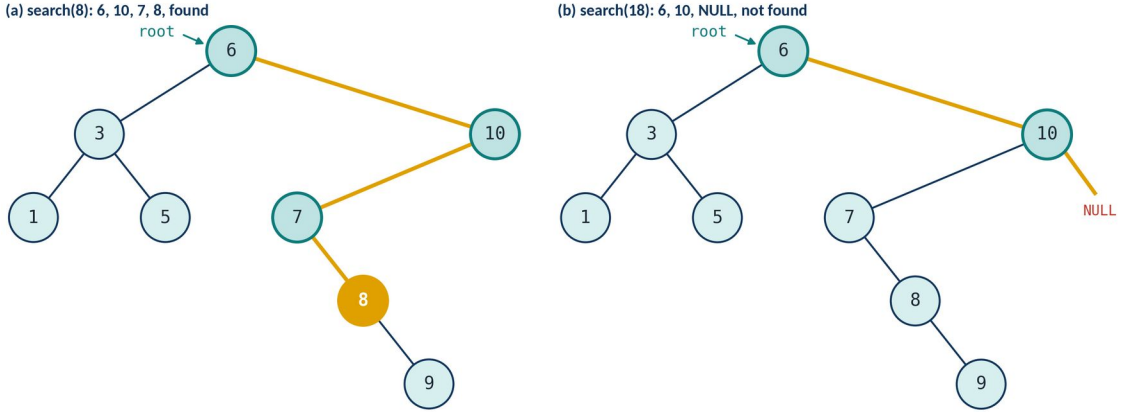


Figure 11.16 Two searches: (a) `search(8)` follows 6, 10, 7 and finds 8; (b) `search(18)` follows 6 and 10, falls off the tree at the empty right subtree of 10, and reports that 18 is absent.

Figure 11.16: `search(8)` compares with 6, 10, 7, 8: four comparisons in eight nodes; `search(18)` proves absence after two. Book: Section 11.7.3 and Figure 11.16.

Minimum, Maximum, Height

Pseudocode 11.7 The minimum is the leftmost node and the maximum the rightmost node of a nonempty tree.

```
TREE-MINIMUM(x):
 1.  while x.left ≠ NIL
 2.      x ← x.left
 3.  return x
```

```
TREE-MAXIMUM(x):
 1.  while x.right ≠ NIL
 2.      x ← x.right
 3.  return x
```

$$height(\text{NIL}) = -1, height(x) = 1 + \max(height(x.left), height(x.right)) \tag{11.9}$$

Pseudocode 11.7: leftmost and rightmost node, $O(h)$. Equation (11.9) gives height 2 for the running example; counting nodes and leaves is the same postorder pattern, $\Theta(n)$. Printing sideways is a reverse inorder. Book: Sections 11.7.4–11.7.6.

The Leftmost and the Rightmost

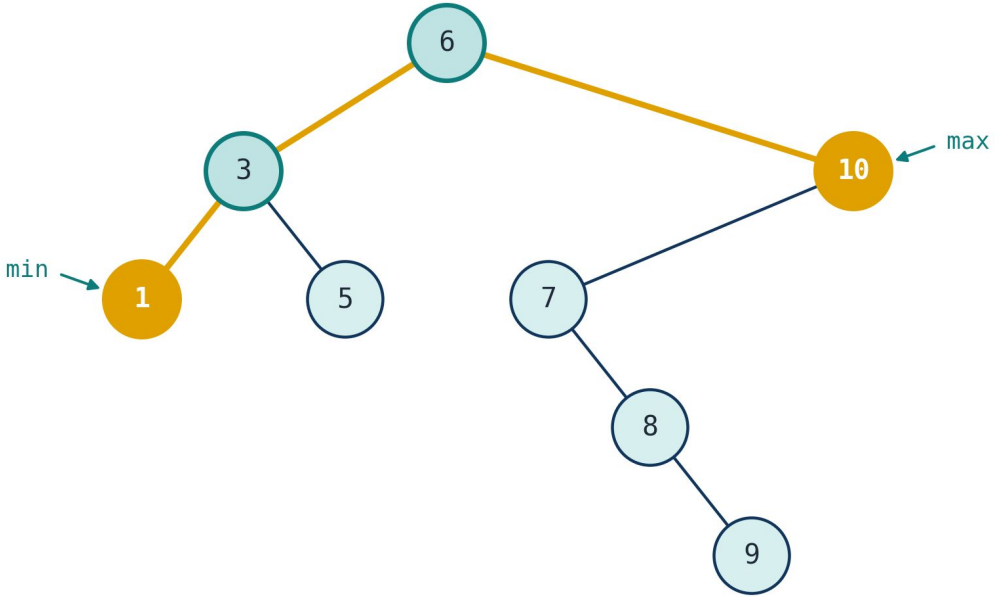


Figure 11.17 The minimum, 1, is reached by going left from the root until there is no left child; the maximum, 10, by going right until there is no right child.

Figure 11.17: the maximum, 10, is not a leaf; it only has no right child. Book: Section 11.7.4 and Figure 11.17.

Successor: Two Cases

Right subtree exists: the successor is its minimum.

No right subtree: the successor is the last node where the search path from the root *turned left*.

Pseudocode 11.8 `TREE-SUCCESSOR` without parent pointers: search for k while remembering the last left turn.

```
TREE-SUCCESSOR(root, k):              // k is in the tree
 1.  cand ← NIL                       // last node where the path turned left
 2.  x ← root
 3.  while x.key ≠ k
 4.      if k < x.key
 5.          cand ← x
 6.          x ← x.left
 7.      else x ← x.right
 8.  if x.right ≠ NIL
 9.      return TREE-MINIMUM(x.right)  // case 1
10.  return cand                      // case 2 (NIL if k is the maximum)
```

Pseudocode 11.8 walks down remembering the last left turn, because our nodes have no parent pointer. The predecessor is the mirror image. A whole walk by successors costs $\Theta(n)$: that is `++it` of `std::map`. Book: Section 11.7.9.

The Two Cases, Drawn

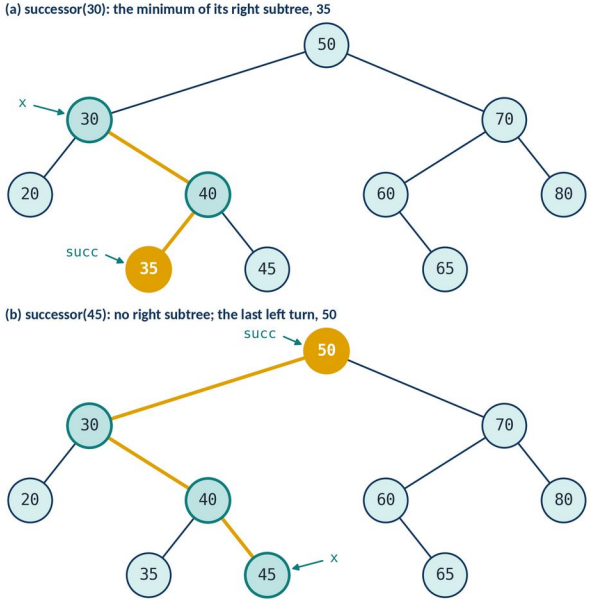


Figure 11.18 The two cases of the successor: (a) 30 has a right subtree, and its successor is that subtree's minimum, 35; (b) 45 has no right subtree, and its successor is the ancestor at which the path from the root last turned left, 50.

Figure 11.18: inorder 20, 30, 35, 40, 45, 50, 60, 65, 70, 80; successor(30) = 35, successor(45) = 50. Book: Section 11.7.9 and Figure 11.18.

Deletion: Three Cases

Case 1, a leaf: set the parent's pointer to NIL and release the node.

Case 2, one child: splice the node out; the child moves up with its whole subtree.

Case 3, two children (Hibbard): copy the successor's key into the node, then delete the successor from the right subtree; it has no left child, so that is case 1 or 2.

The BST property survives each case; the inorder sequence simply loses the deleted key. Book: Section 11.7.10.

Worked Example: Deleting 10, 5, 65

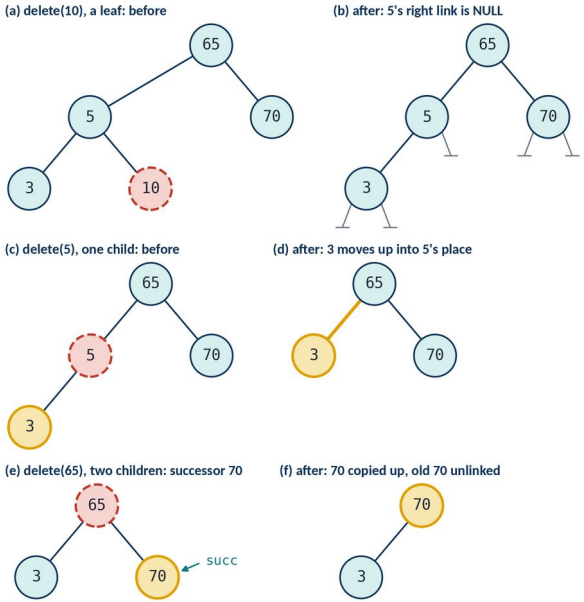


Figure 11.19 The three cases of deletion, applied one after the other to the tree of `insert(65, 5, 70, 3, 10)`: a leaf is unlinked (a, b); a node with one child is replaced by that child (c, d); a node with two children takes the key of its successor, which is then deleted from the right subtree (e, f).

Figure 11.19: cases 1, 2 and 3 in turn; for 65 the key 70 is copied up. Book: Section 11.7.10 and Figure 11.19.

TREE-DELETE

Pseudocode 11.9 TREE-DELETE, successor variant: a leaf is removed, a node with one child is replaced by the child, and a node with two children takes its successor's key.

```
TREE-DELETE(x, k):                  // returns the root of the subtree after deleting k
 1.  if x = NIL
 2.      return NIL                 // k is not in the tree
 3.  if k < x.key
 4.      x.left ← TREE-DELETE(x.left, k)
 5.  else if k > x.key
 6.      x.right ← TREE-DELETE(x.right, k)
 7.  else if x.left = NIL and x.right = NIL     // case 1: a leaf
 8.      free x; return NIL
 9.  else if x.left = NIL or x.right = NIL      // case 2: one child
10.      c ← the non-NIL child of x
11.      free x; return c
12.  else                                       // case 3: two children
13.      s ← TREE-MINIMUM(x.right)              // the successor
14.      x.key ← s.key
15.      x.right ← TREE-DELETE(x.right, s.key)
16.  return x
```

Pseudocode 11.9 returns the root of the subtree after the deletion, and the caller stores it: no special case for the root. In case 3 the key moves, not the node; libraries relink nodes instead. Book: Sections 11.7.10 and 11.7.12.

Successor or Predecessor?

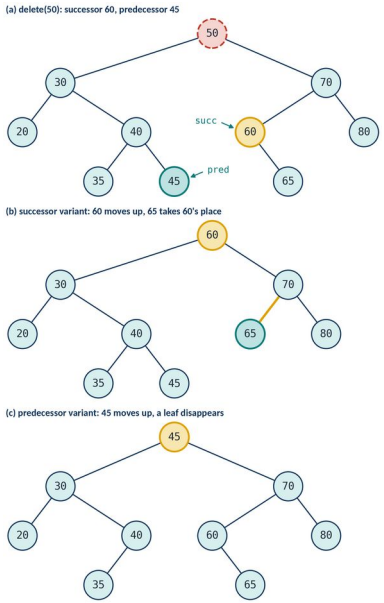


Figure 11.20 Deleting a node with two children, both ways: (a) the root 50 has successor 60 and predecessor 45; (b) with the successor, 60 moves up and its right child 65 takes its old place (case 2); (c) with the predecessor, 45 moves up and a leaf disappears (case 1).

Figure 11.20: both results are correct. Always using the successor makes trees measurably taller over long runs; alternate, or choose at random. Book: Section 11.7.11 and Figure 11.20.

Every Operation Costs $O(h)$

$$T_{\text{search}}, T_{\text{insert}}, T_{\text{delete}}, T_{\text{min}}, T_{\text{max}}, T_{\text{succ}}, T_{\text{pred}} = O(h) \tag{11.10}$$

Table 11.2 The cost of the operations of a binary search tree with n nodes and height h.

Operation	Cost	Balanced tree, $h=\Theta(\lg n)$	Degenerate tree, $h=n-1$
search, insert, delete	$O(h)$ (Pseudocode 11.6, Pseudocode 11.5, Pseudocode 11.9)	$O(\lg n)$	$O(n)$
minimum, maximum	$O(h)$ (Pseudocode 11.7)	$O(\lg n)$	$O(n)$
successor, predecessor	$O(h)$ (Pseudocode 11.8)	$O(\lg n)$	$O(n)$
count, leaves, height	$\Theta(n)$ (Equation (11.9))	$\Theta(n)$	$\Theta(n)$
any traversal	$\Theta(n)$ time (Equation (11.8)), $O(h)$ stack	$\Theta(n)$	$\Theta(n)$
build from n keys	n insertions	$O(n \lg n)$	$\Theta(n^2)$

Equation (11.10) and Table 11.2: a BST is as good as binary search when its height is logarithmic, and no better than a linked list when its height is linear. Book: Section 11.7.13.

The Order of Insertion Decides the Shape

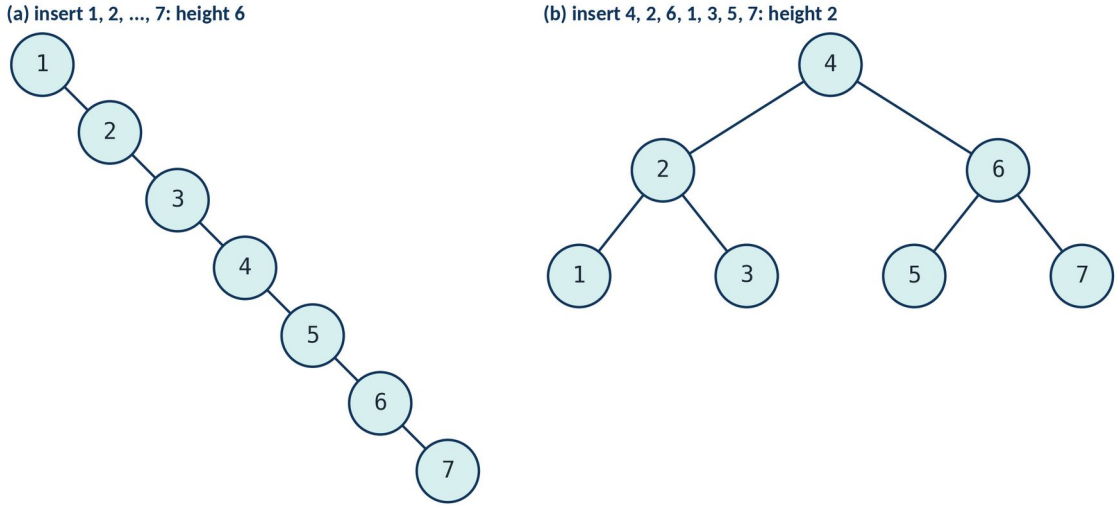


Figure 11.21 The same seven keys in two insertion orders: (a) in sorted order they form a chain of height 6; (b) inserted middle first they form a perfect tree of height 2.

Figure 11.21: sorted input builds a chain of height 6; 4, 2, 6, 1, 3, 5, 7 builds a perfect tree of height 2. By Equation (11.11), 64 of the 5,040 orders of seven keys give a chain. Book: Section 11.8.1 and Figure 11.21.

Measured Heights

$$\#\left\{\text{orders giving height } n-1\right\}=2^{n-1} \tag{11.11}$$

$$\acute{d}(n)=2\left(1+\frac{1}{n}\right)H_n-4\approx 2\ln n\approx 1.39\lg n \tag{11.12}$$

Random order: a million keys give height 51 and average depth 24.83, matching Equation (11.12).

Sorted order: height exactly $n-1$, and the build time grows fourfold per doubling: 806 ms for 32,000 keys against 3.44 ms at random.

Sorted input is the worst case and, in practice, a common one: records arrive in order of creation. In Python the sorted build of 32,000 keys took about 16 seconds. Book: Section 11.8.2 and Equation (11.12).

Random versus Sorted, Plotted

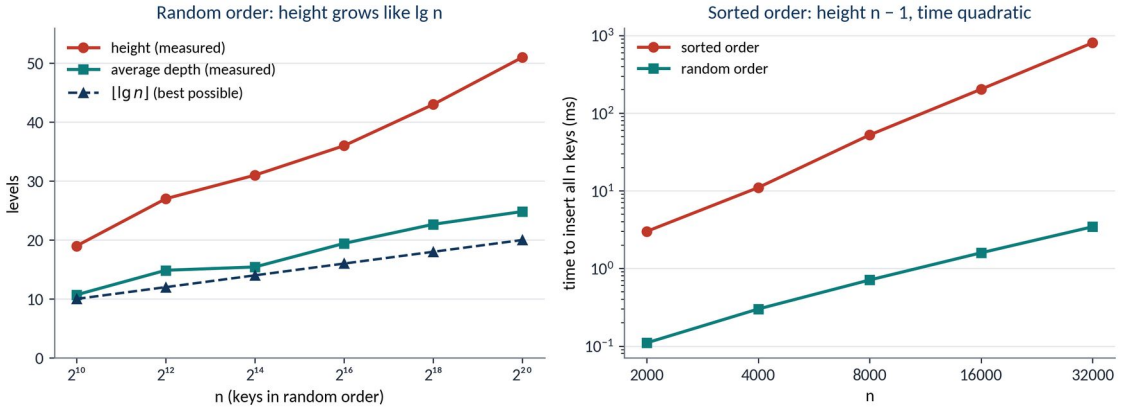


Figure 11.22 The measurements: in random order the height and the average depth grow in step with $\lg n$ (left); in sorted order the tree is a chain, and the time to build it grows fourfold with every doubling of n (right, logarithmic scale).

Figure 11.22: the heights are fixed by the seed; only the times depend on the machine. Book: Section 11.8.2 and Figure 11.22.

Rotations

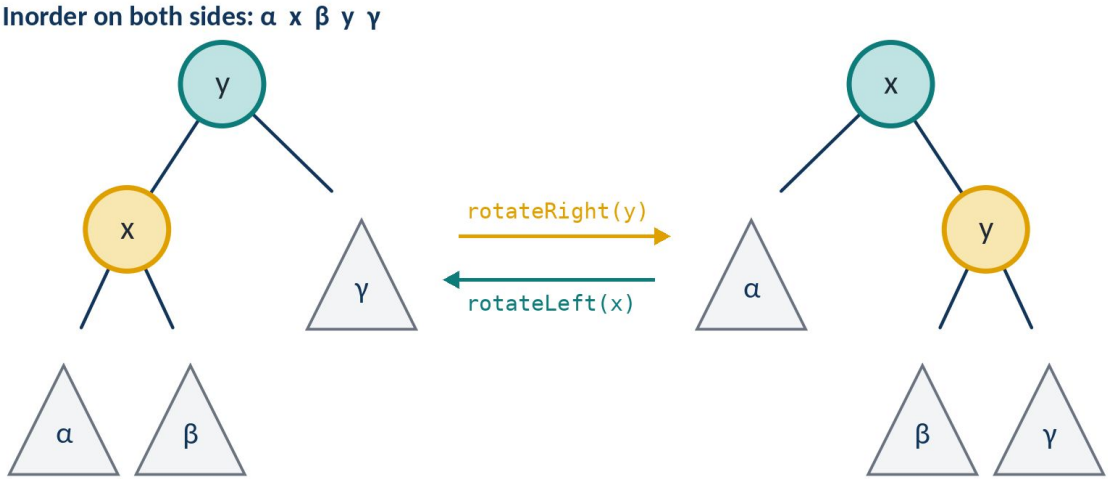


Figure 11.23 Right and left rotation: three pointers change, the subtree rooted at the top becomes one level lower on one side and one level higher on the other, and the inorder sequence $\alpha, x, \beta, y, \gamma$ is unchanged.

Figure 11.23: three pointers change, $O(1)$ time, and the inorder sequence $\alpha, x, \beta, y, \gamma$ is unchanged, so the BST property survives. Book: Section 11.9.1 and Figure 11.23.

AVL Trees

> **DEFINITION.** A binary tree is *height-balanced* if, at every node, the heights of the two subtrees differ by at most 1: the *balance factor* is -1, 0 or $+1$ everywhere.

AVL (1962) keeps every node height-balanced, repairing a factor of ± 2 with one rotation, or two when the deep part is on the inner side.

$$h < 1.4405 \lg(n+2) - 0.3277 \tag{11.13}$$

The sparsest AVL tree of height h has $N(h) = N(h-1) + N(h-2) + 1$ nodes, the Fibonacci recurrence of Chapter 10, hence Equation (11.13): never more than 44% taller than the best tree. A million sorted keys: height 19. Book: Sections 11.8.3 and 11.9.2.

Sorted Keys into an AVL Tree

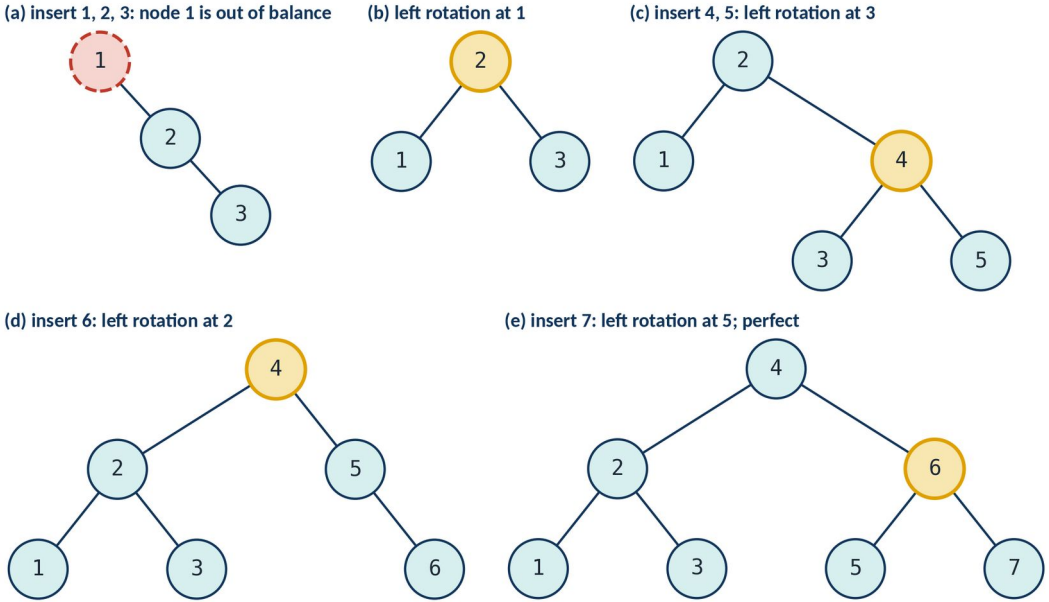


Figure 11.24 Inserting 1 to 7 in sorted order into an AVL tree: (a) after 3, node 1 has balance factor −2; (b) a left rotation at 1 repairs it; (c) to (e) three more left rotations keep the tree balanced, and the result is perfect.

Figure 11.24: four left rotations, and 1 to 7 end in a perfect tree; `avl_first_look.cpp` prints the same rotations. Book: Section 11.9.2 and Figure 11.24.

Red-Black Trees

1. Every node is red or black. **2.** The root is black. **3.** Every NIL leaf is black. **4.** A red node has two black children. **5.** Every path from a node to NIL has the same number of black nodes.

$$h \leq 2 \lg(n+1) \tag{11.14}$$

Weaker than AVL, but at most 2 rotations per insertion and 3 per deletion: `std::map`, `std::set`, `TreeMap`, `TreeSet` and the Linux kernel use it.

Rules 4 and 5 make the longest path at most twice the shortest, hence Equation (11.14). AVL trees, being lower, suit search-heavy work. Book: Section 11.9.3 and Equation (11.14).

A Red-Black Coloring

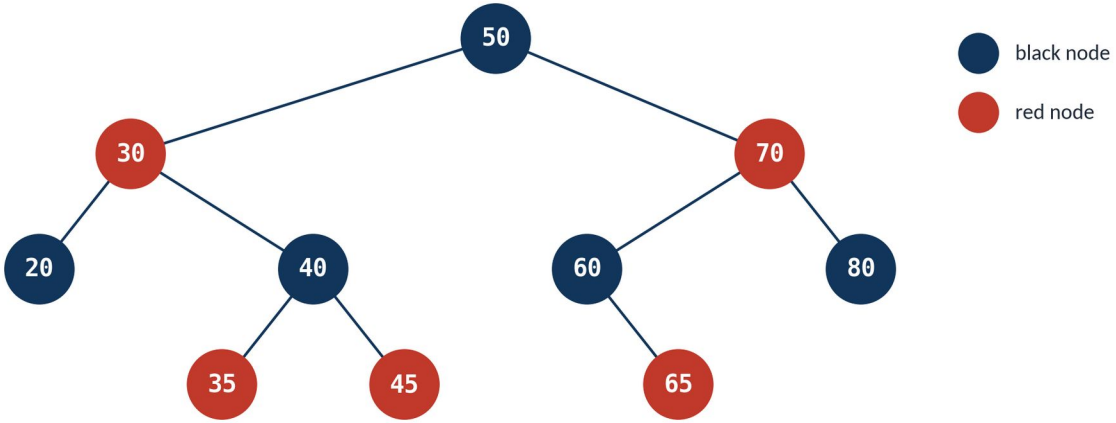


Figure 11.25 The tree of Figure 11.18 colored as a red-black tree: the root is black, red nodes have black children, and every root-to-NIL path contains two black nodes.

Figure 11.25: every root-to-NIL path passes exactly two black nodes, and no red node has a red child. B-trees, splay trees and treaps complete the family. Book: Sections 11.9.3–11.9.4 and Figure 11.25.

First Child, Next Sibling

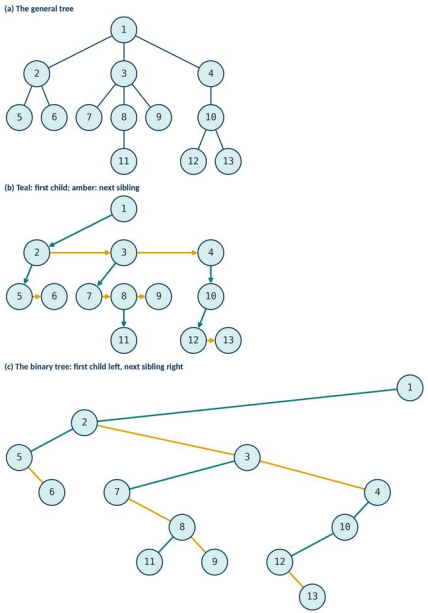


Figure 11.26 Converting a general tree to a binary tree: (a) the general tree; (b) each node keeps only a link to its first child (teal) and to its next sibling (amber); (c) the same links drawn as a binary tree, first child on the left and next sibling on the right.

Figure 11.26: first child to the left, next sibling to the right. General preorder = binary preorder; general postorder = binary inorder. Book: Section 11.10.1 and Figure 11.26.

Worked Example: Preorder plus Degrees

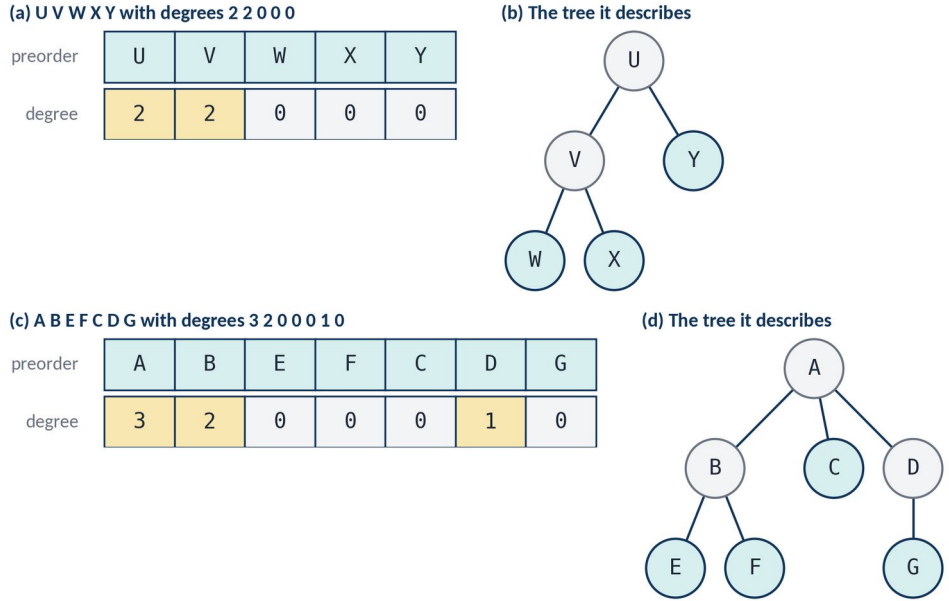


Figure 11.27 Two trees rebuilt from their preorder listing and the degree of each node: U V W X Y with degrees 2 2 0 0 0 (a, b), and A B E F C D G with degrees 3 2 0 0 0 1 0 (c, d).

Figure 11.27: scan right to left; a node of degree d takes the d subtrees to its right. U V W X Y with 2 2 0 0 0 gives U(V(W,X),Y); the degrees always sum to $n-1$. Book: Sections 11.10.2–11.10.3 and Figure 11.27.

Search Trees in the Libraries

Table 11.3 Ordered sets and maps in the standard libraries of C++, Java and Python, and their hash-based alternatives.

Need	C++	Java	Python
Ordered set	`std::set` (red-black)	`TreeSet` (red-black)	sorted `list` + `bisect`; `SortedSet` (third party)
Ordered map	`std::map` (red-black)	`TreeMap` (red-black)	`SortedDict` (third party)
Successor, predecessor	`std::next`, `std::prev`, `upper_bound`	`higherKey`, `lowerKey`	`bisect_right`, `bisect_left`
Range query	`lower_bound` .. `lower_bound`	`subMap`	slice between two `bisect_left` calls; `irange`
Unordered (hash)	`std::unordered_map`	`HashMap`	`dict`

Table 11.3: iterating a `std::map` is an inorder walk; `lower_bound` and `subMap` answer range queries. Python has no built-in search tree: a sorted list with `bisect`, or `sortedcontainers`. Book: Section 11.11.

Profitina in Practice

A web page is a tree (the DOM); document order is its preorder. **Headings form an outline**; checking it is a walk with a stack of open headings.

Structured data is a tree (JSON-LD). **A site is a tree of links**: level order assigns each page its crawl depth.

Ordered data lives in B-trees: "scans between July and September" is a range query on a PostgreSQL B-tree index.

Listing 11.2 The pruned range query of `laundry_tree.cpp`, Profitina Laundry's pickup board keyed by ready time.

```
// Orders ready in [lo, hi], visiting only subtrees that can hold such times.
void range(const Node *t, int lo, int hi, int &visited) {
    if (t == nullptr) return;
    ++visited;
    if (lo < t->order.ready) range(t->left, lo, hi, visited);
    if (lo <= t->order.ready && t->order.ready <= hi)
        std::printf("  %s  #%d %s\n", hhmm(t->order.ready).c_str(), t->order.number,
                    t->order.customer.c_str());
    if (t->order.ready < hi) range(t->right, lo, hi, visited);
}
```

Listing 11.2 goes left only if `lo` is smaller than the node's time and right only if the time is smaller than `hi`. Book: Section 11.12.

The Pickup Board



Figure 11.28 The pickup board's tree, keyed by ready time: the range query for 10:00 to 12:59 visits five nodes (teal and amber) and reports three (amber); the other seven are never touched.

Figure 11.28: the query visits 5 of 12 nodes. The height is 7, not 3: ready times arrive nearly sorted, so a real system would use `std::map` *or* `TreeMap`*. Book: Section 11.12 and Figure 11.28.*

Common Mistakes

Confusing depth and height; **checking the BST property only between parent and children** (10(5(2, 12), 15) is not a BST).

Losing the change to a pointer: `void insert(Node *root, int x)` changes a copy; forgetting `root = insert(root, x)` in Java or Python.

Splicing a node with two children; **dereferencing NIL**; **releasing in the wrong order** (delete children first: postorder).

Assuming $O(\lg n)$ for a plain BST, and **recursing on a tree that may be degenerate**.

And: using the wrong traversal for the job, and mixing up the names of shapes. Book: Section 11.13.

Summary: Five Things to Remember

1. A tree is a root with disjoint subtrees; n nodes, $n-1$ edges; height at least $\lfloor \lg n \rfloor$ for a binary tree.

2. Preorder, inorder, postorder and level order all take $\Theta(n)$; postfix builds an expression tree with a stack.

3. In a BST, inorder is sorted; insert, search, minimum, successor and delete all cost $O(h)$.

4. Deletion: unlink a leaf, splice one child, borrow the successor's or predecessor's key for two.

5. Sorted input makes a chain; rotations keep AVL and red-black trees logarithmic, and the libraries use red-black trees.

Book: Section 11.14.

Next Time

Chapter 12 — Evaluation and Discussion III. Recursion at work on lists and trees: problems solved from a blank page, in C++, Java and Python.

Trees built exactly like today's, checked by programs.

Before then: work Exercises 11.1, 11.5, 11.8 and 11.10.

The solutions, with programs, are in the book. Book: Sections 11.15 and 11.16.

Lecture 12 — Evaluation and Discussion III: Recursion at Work on Lists and Trees

Course: Data Structure (EF234201)

Book: Irfan Subakti, *Data Structure with C++, Java and Python*, Chapter 12

Lecturer: Irfan Subakti — yifana@gmail.com

No new structure today: sixteen problems on lists and trees, each solved from a blank page, checked by a program, and set beside the tempting wrong solution. Book: Chapter 12, Sections 12.1–12.20.

What This Meeting Covers

1. **A method** — seven steps for a recursive problem on a linked structure, and the template of a tree function.
2. **Problems 12.1–12.6** — depth-first order, one level, the sum, tripling, the mirror, the balance check.
3. **Problems 12.7–12.10** — deletion, the parent, the children, and a tree flattened into a doubly linked list.
4. **Problems 12.11–12.16** — reversing a doubly linked list, the Josephus circle, choosing books, checking a BST, the lowest common ancestor, printing level by level.

Every problem follows the same path: the statement, in our own words, by hand, the solution in C++ (with Java and Python in the book), a tempting wrong solution, and a check-yourself question. Book: Section 12.1.

Where We Stand: Three Tools

Chapter 9: a second link per node. Walk both ways, remove a node you hold; every forward arrow that changes needs its backward twin.

Chapter 10: recursion. A base case answered directly, a recursive case that trusts the smaller call; the call stack; recurrences.

Chapter 11: trees and binary search trees. Insert, traverse, search, count, height, delete; nearly all of them recursive.

> **THE ONE IDEA.** A tree is empty, or a root with a left tree and a right tree: answer the empty tree directly, ask the same question of the two smaller trees, and combine the two answers with the root.

When a program today goes wrong, one of the three parts is broken: the empty tree answered wrongly, the question changed on the way down, or a smaller tree's answer thrown away instead of combined. Book: Section 12.1.

The Seven-Step Method

Table 12.1 The seven-step method for a recursive problem on a list or a tree. Steps 1 and 2 pin down the answer, Steps 3 to 5 design the recursion, and Steps 6 and 7 check it.

Step	Write on paper	Question it answers
1	The problem in your own words, and one example by hand	Do I know what the output must be?
2	The answer for the empty tree or list	What is the base case, and is it right for one node?
3	The tree, with the arguments each call receives	Does every child receive the right question?
4	Root, left answer, right answer, and the order of the three	How do the answers combine, and when is the node visited?
5	`root->left = f(root->left)` wherever links change	Does every parent re-attach what its child returns?
6	The edge cases	Empty, one node, one child, absent value, a long chain?
7	Tests, then the same tests under a memory checker	Does it work, and does it only *seem* to work?

Table 12.1: Steps 1 and 2 pin down the answer, Steps 3 to 5 design the recursion, Steps 6 and 7 check it. Every C++ program was also run under AddressSanitizer and Valgrind. Book: Section 12.2.

The Template of a Tree Function

Pseudocode 12.1 The template of a recursive tree function. Filling the blanks with 0 and $T.key+a+b$ gives the sum of Problem 12.3; with -1 and $1+\max(a,b)$ it gives the height.

```
TREE-FUNCTION(T)
1  if T = NIL
2      return ANSWER-FOR-THE-EMPTY-TREE          // Step 2: the base case
3  a = TREE-FUNCTION(T.left)                     // trust the smaller call
4  b = TREE-FUNCTION(T.right)                    // trust the smaller call
5  return COMBINE(T.key, a, b)                   // Step 4: root, left, right
```

A tree of n nodes has n calls on non-empty trees and $n+1$ on empty children, so Equation (12.1):

$$T(n)=cn+c_0(n+1)=\Theta(n). \tag{12.1}$$

Pseudocode 12.1: fill the blanks with 0 and $T.key+a+b$ for the sum, with -1 and $1+\max(a,b)$ for the height. Exceptions to linear cost: a walk that stops early, one path of a BST, a walk inside a walk. Book: Section 12.2.

The Tree We Work On

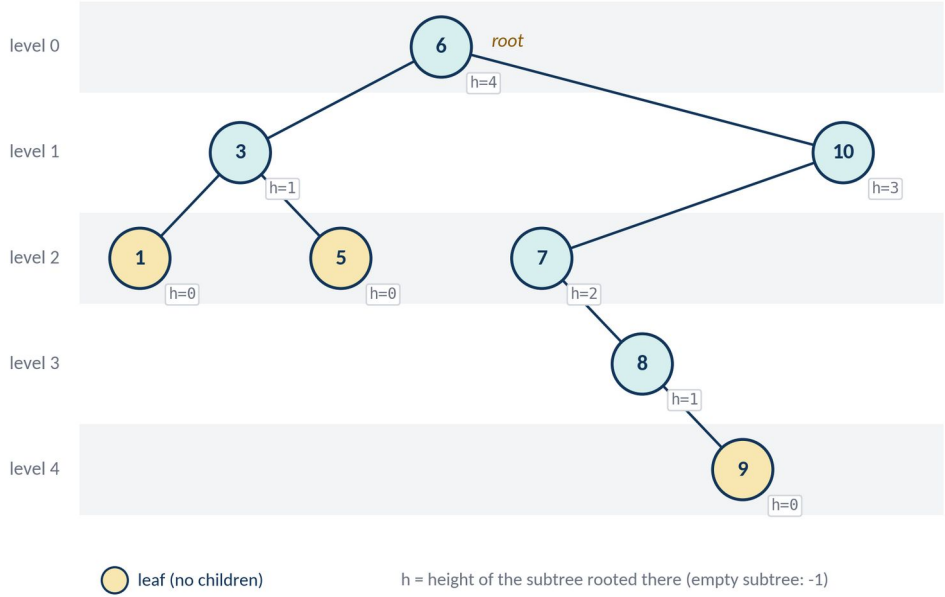


Figure 12.1 The tree of this chapter, built by inserting 6, 3, 10, 1, 5, 7, 8, 9 into an empty binary search tree. The levels are counted from 0 at the root, the leaves are shaded, and h is the height of the subtree rooted at each node.

Figure 12.1: insert 6, 3, 10, 1, 5, 7, 8, 9. Levels 0 to 4, height 4, leaves 1, 5 and 9. Problems 12.1 to 12.10 all walk this picture. Book: Section 12.3.1 and Figure 12.1.

Three Facts We Keep Using

$$h(\varnothing) = -1, h(T) = 1 + \max(h(T.left), h(T.right)). \tag{12.2}$$

$$\text{for every node } x: y < x.key \text{ for every } y \text{ in } x.left,\ y > x.key \text{ for every } y \text{ in } x.right. \tag{12.3}$$

$$n \le 2^{h+1} - 1 \Rightarrow h \ge \lceil \lg(n+1) \rceil - 1, \text{and always } h \le n - 1. \tag{12.4}$$

Equation (12.2) gives a leaf height $1 + \max(-1, -1) = 0$. *Equation (12.3) is about every descendant: 5 is the right child of 3 but must be smaller than 6. By Equation (12.4) our 8 nodes have height between 3 and 7. Book: Section 12.3.1.*

Problem 12.1 — Depth First, Left Branch First

PROBLEM. Write a recursive function to print the depth-first traversal of a tree, i.e., choosing left branches first. `void printDepthFirst(Node *root)`

Listing 12.1 `printDepthFirst` from `p1201_depth_first.cpp`: the node, then its whole left subtree, then its whole right subtree.

```
// Visit the node, then everything on its left, then everything on its right.
void printDepthFirst(Node *root) {
    if (root == nullptr) return;         // base case: an empty tree prints nothing
    printf("%d ", root->data);           // 1. the node itself
    printDepthFirst(root->left);         // 2. the whole left subtree
    printDepthFirst(root->right);        // 3. the whole right subtree
}
```

Listing 12.1 is preorder: `6 3 1 5 10 7 8 9`*. The book also gives a version with an explicit* `std::stack`*, pushing the right child before the left.* $\Theta(n)$ *time,* $O(h)$ *stack. Book: Sections 12.4.1–12.4.4.*

The Order of the Prints

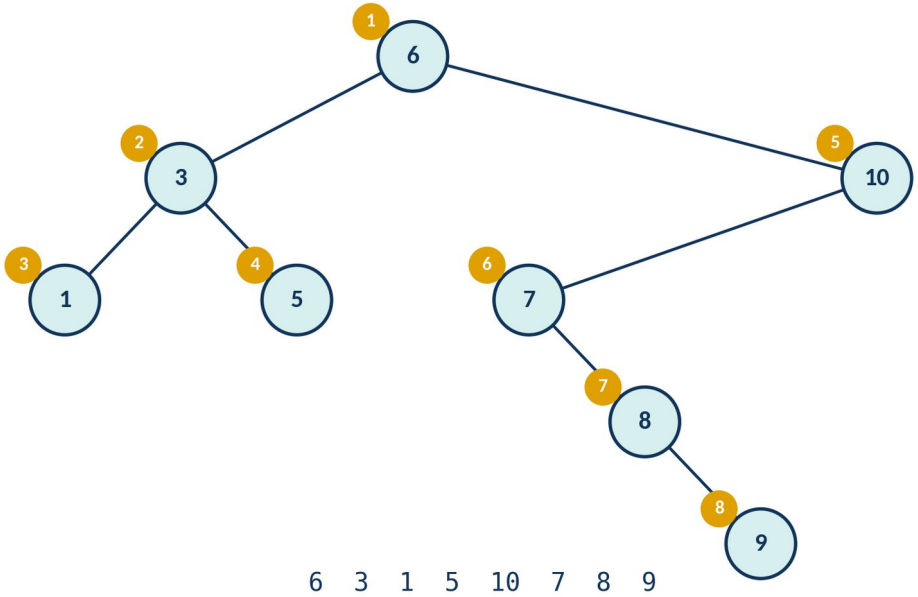


Figure 12.2 The depth-first order of Problem 12.1: each badge gives the moment the node is printed. The whole left subtree of 6 (badges 2 to 4) is finished before the right subtree (badges 5 to 8) is entered.

Figure 12.2: when 9 is printed, the calls on 6, 10, 7, 8 and 9 are all waiting: one more than the height. Book: Section 12.4.3 and Figure 12.2.

Wrong: No Base Case

```
void printDepthFirst(Node *root) {
    printf("%d ", root->data);          // WRONG: no base case at all
    printDepthFirst(root->left);
    printDepthFirst(root->right);
}
```

GCC warns `infinite recursion detected`; the run prints 6, 3, 1 and then reads `root->data` through the left child of 1, a `nullptr`: *SEGV on unknown address 0x000000000000*.

Read a sanitizer report from the top: *what* happened, *where* (line 7), and *how we got there* (line 8).

Every tree, however large, ends in $n+1$ `nullptr` children, so every recursive tree function must say what happens at `nullptr`. Book: Section 12.4.6.

Problem 12.2 — Printing One Level

PROBLEM. Write a recursive function to print all nodes at the k-th level in the binary tree. `void printKthLevel(Node *root, int k)`

Listing 12.2 `printKthLevel` from `p1202_kth_level.cpp`: the question changes on the way down, from k to $k-1$.

```
void printKthLevel(Node *root, int k) {
    if (root == nullptr) return;          // fell off the tree: nothing on this path
    if (k == 0) {                         // this node is exactly k levels down
        printf("%d ", root->data);
        return;
    }
    printKthLevel(root->left, k - 1);     // one level lower, one step fewer to go
    printKthLevel(root->right, k - 1);
}
```

Listing 12.2 tests `root == nullptr` *before* `k == 0`*. Levels 0 to 5 print* `6`*,* `3 10`*,* `1 5 7`*,* `8`*,* `9` *and an empty line. Printing all levels this way costs up to* $\Theta(n^2)$ *on a chain. Book: Sections 12.5.1–12.5.4.*

The k Each Call Receives

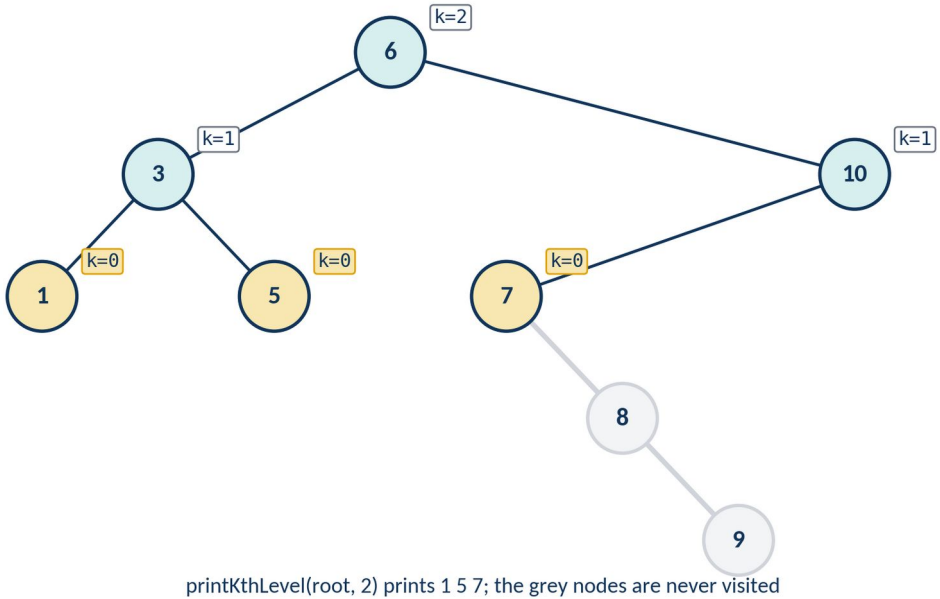


Figure 12.3 The call `printKthLevel(root, 2)`: each node's tag is the k its call receives. The nodes that receive $k = 0$ print themselves and stop, so 8 and 9, below them, are never visited.

Figure 12.3: for level 2, the calls with $k = 0$ print 1, 5 and 7 and stop, so 8 and 9 are never visited. Book: Section 12.5.3 and Figure 12.3.

Wrong: `--k` Changes k Itself

```
    printKthLevel(root->left, --k);      // WRONG: --k changes k itself,
    printKthLevel(root->right, --k);     // so the right side gets k - 2
```

```
level 0: 6
level 1: 3
level 2: 1 10
level 3: 5 7
```

The first call receives $k-1$, but it has also changed the local k, so the second call receives $k-2$: the right side appears one line early. `k - 1` computes a value and leaves `k` alone. Step 3 of the method shows it at once. Book: Section 12.5.6.

Problem 12.3 — The Sum of All Nodes

PROBLEM. Write a recursive function to calculate the sum of all nodes in the binary tree. `int sumOfNodes(Node *root)`

$$S(\varnothing)=0, S(T)=T.key+S(T.left)+S(T.right). \tag{12.5}$$

Listing 12.3 `sumOfNodes` from `p1203_sum.cpp`: one line of recursion.

```
int sumOfNodes(Node *root) {
    if (root == nullptr) return 0;       // the empty tree adds nothing
    return root->data + sumOfNodes(root->left) + sumOfNodes(root->right);
}
```

Equation (12.5) and Listing 12.3: the base case is 0 because 0 changes nothing when added. The whole tree gives 49, its two subtrees 9 and 34, and $6+9+34=49$. *Book: Sections 12.6.1–12.6.4.*

The Values the Calls Return

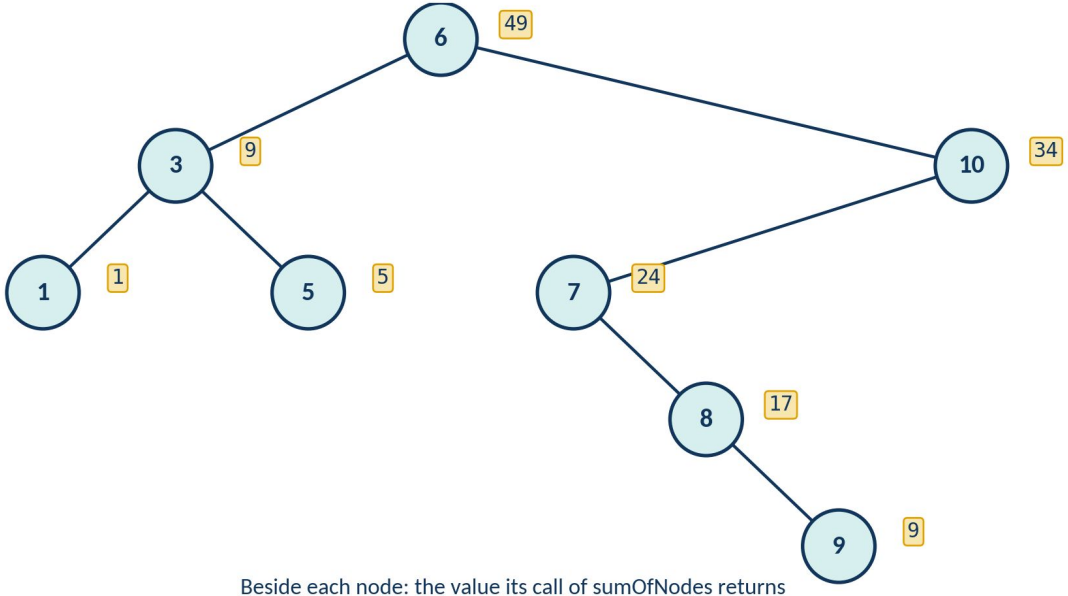


Figure 12.4 The values returned by `sumOfNodes` on each subtree. Each is the node's own value plus the two values returned below it; the root returns 49.

Figure 12.4: read bottom up, the order in which the calls finish: 8 returns 17, 7 returns 24, 10 returns 34, 3 returns 9, the root 49. Book: Section 12.6.3 and Figure 12.4.

Wrong: a Running Total Outside

```
int total = 0;                  // WRONG: a running total outside the function

int sumOfNodes(Node *root) {
    if (root == nullptr) return total;
    total += root->data;
    sumOfNodes(root->left);
    sumOfNodes(root->right);
    return total;
}
```

```
sum of the whole tree  : 49
sum again (same answer): 98
left subtree of 6      : 107
```

The first call is right, which makes the mistake tempting; `total` remembers every earlier call. A recursive function combines what its calls return; it does not communicate through shared variables. Book: Section 12.6.6.

Problem 12.4 — Tripling Every Node

PROBLEM. Write a recursive function which multiplies by three every node of the tree. `void triple(Node *root)`

Listing 12.4 `triple` from `p1204_triple.cpp`: the node is changed through the pointer.

```
void triple(Node *root) {
    if (root == nullptr) return;
    root->data *= 3;  // change the node itself, through the pointer
    triple(root->left);
    triple(root->right);
}
```

$$a<b \Rightarrow 3a<3b. \tag{12.6}$$

Listing 12.4; by Equation (12.6) the result is still a BST, and `search` *finds 24. The wrong version* `void triple(Node node)` *triples only copies: nothing changes. Book: Sections 12.7.1–12.7.5.*

Before and After Tripling

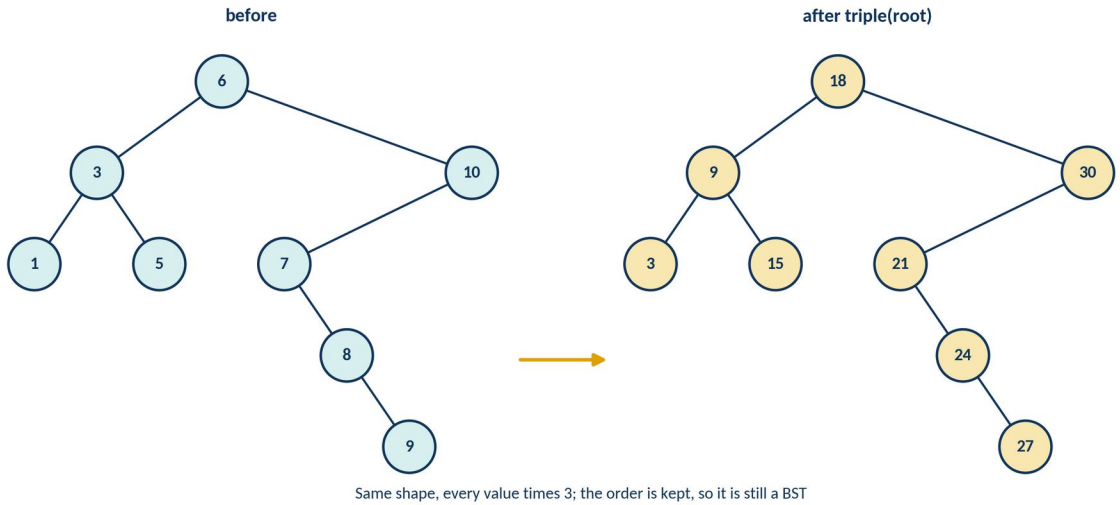


Figure 12.5 Problem 12.4: the tree before and after `triple(root)`. The shape does not change, and since $a<b$ implies $3a<3b$, the tripled tree is still a binary search tree.

Figure 12.5: preorder `6 3 1 5 10 7 8 9` becomes `18 9 3 15 30 21 24 27`; the inorder is still increasing. Book: Section 12.7.2 and Figure 12.5.

Problem 12.5 — The Mirror Image

PROBLEM. Write a recursive function to create a mirror image of the binary tree. `void mirror(Node *root)`

Listing 12.5 `mirror` from `p1205_mirror.cpp`: swap the two subtrees with a saved pointer, then mirror each.

```
void mirror(Node *root) {
    if (root == nullptr) return;
    Node *saved = root->left;            // keep the left subtree reachable
    root->left = root->right;
    root->right = saved;                 // the two subtrees have changed places
    mirror(root->left);                  // now mirror each of them
    mirror(root->right);
}
```

Listing 12.5: preorder `6 10 7 8 9 3 5 1`*, inorder* `10 9 8 7 6 5 3 1`*.* `search(3)` *now fails: the mirror is not a BST. Mirroring twice gives the original back. Book: Sections 12.8.1–12.8.4.*

The Tree in the Mirror

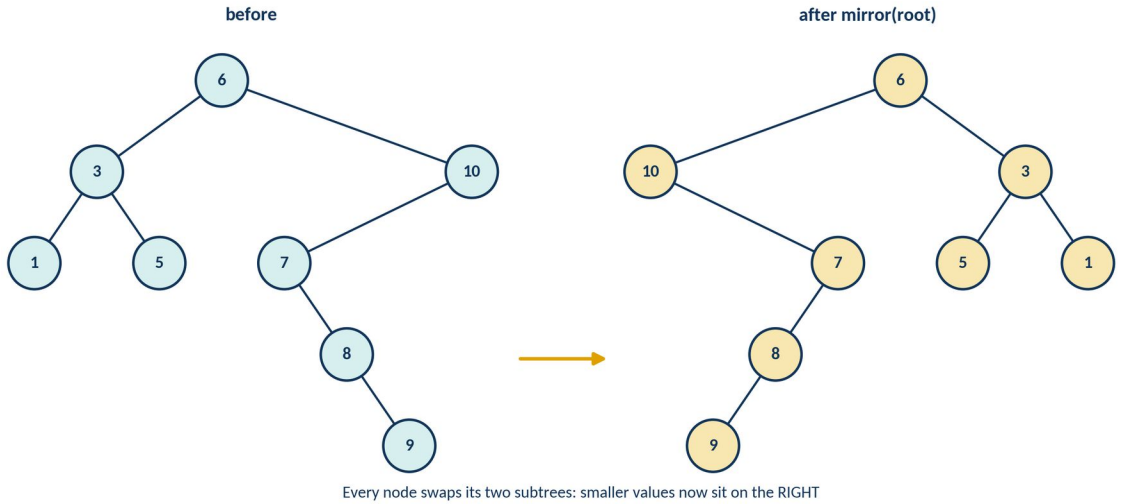


Figure 12.6 Problem 12.5: the tree before and after `mirror(root)`. Every node has swapped its two subtrees, so the smaller values now hang on the right; the result is no longer a binary search tree.

Figure 12.6: every node has swapped its two subtrees; the smaller values now hang on the right. Book: Section 12.8.3 and Figure 12.6.

Wrong: Swapping Without Saving

```
root->left = root->right;            // WRONG: the old left subtree is lost
root->right = root->left;            // ...and this copies the NEW left pointer
```

After the first line both pointers point to the right subtree; the second copies it back. The printed “tree” is `6 10 10`.

Six nodes are unreachable (a leak), and `clear` frees node 10 twice: AddressSanitizer reports *heap-use-after-free*.

The rule of Chapter 8 in a new place: before overwriting a pointer, save what it points to. Python’s tuple assignment saves it for you. Book: Section 12.8.6.

Problem 12.6 — Is the Tree Height-Balanced?

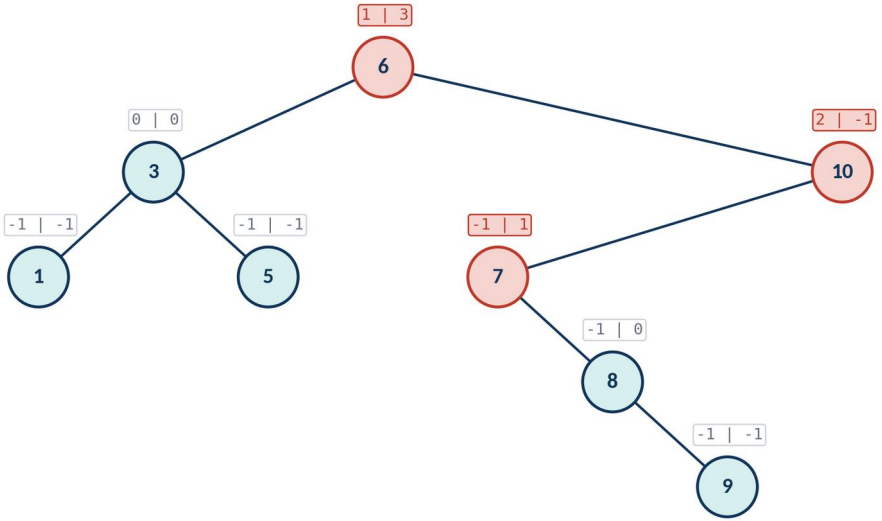
DEFINITION. A binary tree T is *height-balanced* if T is empty, or if $\left|h(T.left) - h(T.right)\right| \le 1$ and both subtrees are height-balanced.

Listing 12.6 `isBalancedHelper` from `p1206_balanced.cpp`: the node, then both subtrees.

```
int isBalancedHelper(Node *root) {
    if (root == nullptr) return 1;        // the empty tree is balanced
    int diff = abs(height(root->left) - height(root->right));
    if (diff > 1) return 0;               // this node breaks the rule
    return isBalancedHelper(root->left) && isBalancedHelper(root->right);
}
```

Listing 12.6: every node must be balanced, not only the root; the empty tree is balanced. The tree 5 3 8 1 4 7 9 0 10 11 hides an unbalanced node 8 below a balanced root. Book: Sections 12.9.1–12.9.4.

Heights Above Each Node



Tag above each node: height(left) | height(right). Red: the two differ by more than 1

Figure 12.7 Problem 12.6: above each node, the height of its left subtree and of its right subtree. The three red nodes break the rule, so the tree is not height-balanced.

Figure 12.7: 7 has heights -1 *and 1, 10 has 2 and* -1*, the root 1 and 3: the answer is 0. Book: Section 12.9.3 and Figure 12.7.*

One Pass Instead of a Walk Inside a Walk

Listing 12.7 `checkedHeight` from `p1206_balanced.cpp`: the height, or `UNBALANCED` as soon as a node breaks the rule.

```
// One pass: return the height, or UNBALANCED as soon as a node breaks the rule.
const int UNBALANCED = -2;              // not -1: -1 already means "empty tree"

int checkedHeight(Node *root) {
    if (root == nullptr) return -1;
    int u = checkedHeight(root->left);
    if (u == UNBALANCED) return UNBALANCED;
    int v = checkedHeight(root->right);
    if (v == UNBALANCED) return UNBALANCED;
    if (abs(u - v) > 1) return UNBALANCED;
    return (u > v ? u : v) + 1;
}
```

$$C = \sum_{x\,\text{checked}} size(x) = \sum_{x\,\text{checked}} (d(x)+1) \le n(h+1), \tag{12.7}$$

Listing 12.7 is $\Theta(n)$; the helper version costs Equation (12.7), $O(n \log n)$ in practice. `UNBALANCED` is -2, because -1 already means "empty tree". Book: Section 12.9.4.

Wrong: “Empty Means Not Balanced”

```
      if (root == nullptr) return 0;       // WRONG: "empty tree: not balanced"
```

```
sample tree            isBalanced = 0, one-pass = 0
6 3 9 1 5 8 10         isBalanced = 0, one-pass = 1
empty tree             isBalanced = 0, one-pass = 1
```

Every tree ends in empty subtrees, so the 0 is carried up by `&&`: even a perfect tree is “not balanced”. It passes the problem’s own example. The base case must change nothing under `&&`: true. Book: Section 12.9.6.

Problem 12.7 — Deleting a Value

PROBLEM. Write a recursive function to delete a particular data. `Node *delNode(Node *root, int delData)`, with the helper `Node *minValueNode(Node *node)`.

Table 12.2 The three cases of deleting a node from a binary search tree. Case 3 is reduced to case 1 or case 2, because the successor never has a left child.

Case	The node to delete has	What takes its place	Why the tree stays a BST
1	no children (a leaf)	nothing: its parent's pointer becomes `nullptr`	no other node moves
2	exactly one child	that child, with its whole subtree	the subtree was already on the correct side of every ancestor
3	two children	its *successor*, the smallest value in its right subtree, copied up; then the successor is deleted from the right subtree	the successor is larger than everything on the left and smaller than everything else on the right

Table 12.2: the function returns the root after the deletion, because the root itself may disappear. Case 3 is Hibbard's scheme of Section 11.7.10. Book: Sections 12.10.1–12.10.2.

The Three Cases on Our Tree

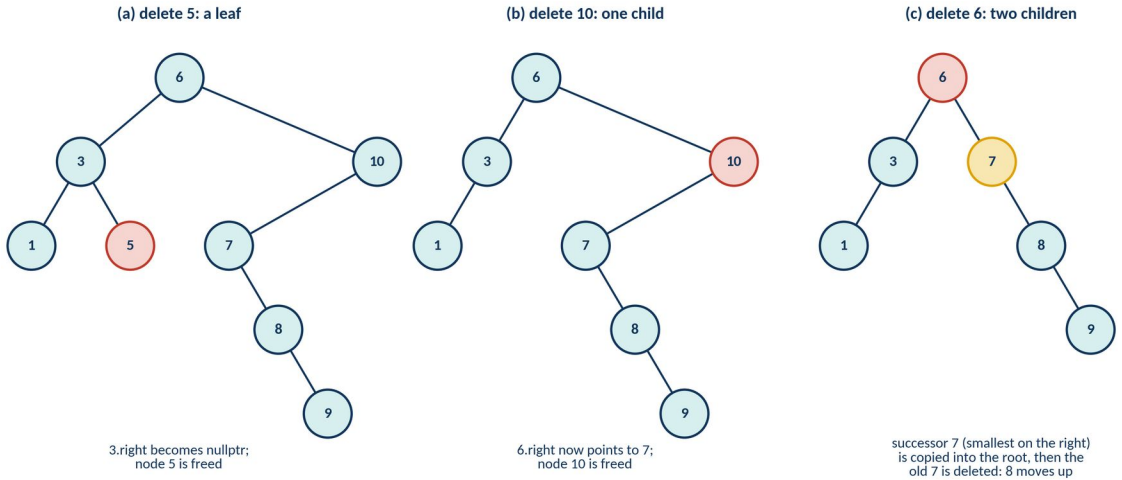


Figure 12.8 The three cases of Problem 12.7 on our tree: (a) the leaf 5 is simply removed; (b) 10 has one child, and 7 takes its place; (c) the root 6 has two children, so its successor 7 is copied into the root and the old 7 is deleted below.

Figure 12.8: delete 5 (a leaf), then 10 (one child, 7 moves up), then 6 (two children: 7 is copied into the root). Deleting 42 changes nothing. Book: Section 12.10.3 and Figure 12.8.

`delNode` in C++

Listing 12.8 `delNode` from `p1207_delete.cpp`: every caller re-attaches what the call returns.

```
Node *delNode(Node *root, int delData) {
    if (root == nullptr) return nullptr;              // not found: nothing changes
    if (delData < root->data) {
        root->left = delNode(root->left, delData);    // re-attach the left side
    } else if (delData > root->data) {
        root->right = delNode(root->right, delData);  // re-attach the right side
    } else if (root->left == nullptr) {               // no left child:
        Node *child = root->right;                    //   the right child (maybe
        delete root;                                  //   nullptr) takes its place
        return child;

    } else {                                          // two children:
        Node *succ = minValueNode(root->right);       //   the next larger value
        root->data = succ->data;                      //   is copied up here,
        root->right = delNode(root->right, succ->data);  // then deleted below
    }
    return root;
}
```

Listing 12.8, with the mirror-image branch for a missing right child omitted: cases 1 and 2 are the same three lines, because in case 1 the only child is `nullptr`*. Preorder after 5, 10, 6:* `7 3 1 8 9`*. Cost* $O(h)$*. Book: Section 12.10.4.*

Wrong: Throwing Away the Result

```
if (delData < root->data) {
    delNode(root->left, delData);         // WRONG: the result is thrown away
} else if (delData > root->data) {
    delNode(root->right, delData);        // WRONG: the result is thrown away
```

The call on 5 frees the node and returns `nullptr`, but 3 ignores it: `3->right` is a *dangling pointer*.

The next traversal reads freed memory: *heap-use-after-free* in `preOrder`, freed in `delNode`, allocated in `insert`.

The fix is one assignment per branch: `root->left = delNode(root->left, delData)`. Mixing `free` with `new` is a second, quieter bug (alloc-dealloc-mismatch). Book: Section 12.10.6.

Problem 12.8 — Finding the Parent

PROBLEM. Write a recursive function to find the parent node of a given child data. `Node *findParent(Node *root, int *childData)`

Listing 12.9 `findParent` from `p1208_find_parent.cpp`: look one step ahead on the search path.

```
Node *findParent(Node *root, int childData) {
    if (root == nullptr || root->data == childData) return nullptr;
    Node *next = (childData < root->data) ? root->left : root->right;
    if (next == nullptr) return nullptr;  // the path ends: not in the tree
    if (next->data == childData) return root;        // root is the parent
    return findParent(next, childData);              // ask the same question below
}
```

Listing 12.9 takes the value by `int`, since it only reads it. Parents: 8→7, 10→6, 9→8, 1→3; 6 is the root and 4 is absent, both `nullptr`. An any-tree version looks on both sides, $\Theta(n)$. Book: Sections 12.11.1–12.11.5.

One Step Ahead on the Path

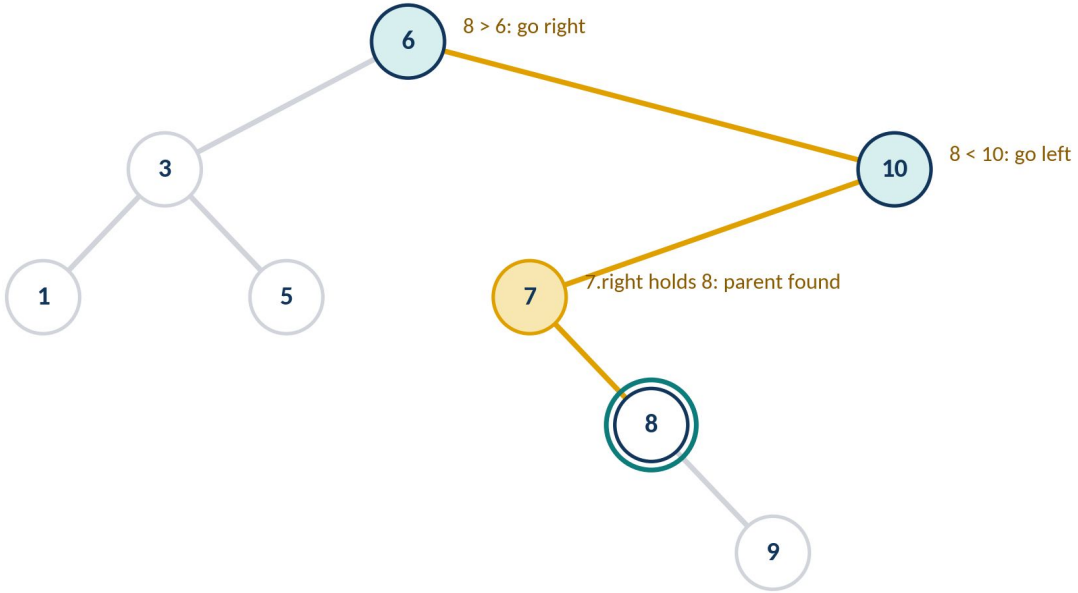


Figure 12.9 `findParent(root, 8)` follows the search path for 8. At every node it looks one step ahead; at 7 the next node on the path holds 8, so 7 is the parent.

Figure 12.9: at 7 the next node holds 8, so 7 is the parent. The wrong one-liner `root->left->data == childData || …` *crashes at 10, whose right child is missing. Book: Sections 12.11.3 and 12.11.6 and Figure 12.9.*

Problem 12.9 — Reporting the Children

Listing 12.10 `findChild` from `p1209_find_child.cpp`: one call of `search` and two careful looks.

```
Node *findChild(Node *root, int parentData) {
    Node *parent = search(root, parentData);   // the recursive walk
    if (parent == nullptr) {
        printf("%d is not in the tree\n", parentData);
        return nullptr;
    }
    if (parent->left != nullptr) {
        printf("left child of %d: %d\n", parentData, parent->left->data);
    } else {
        printf("%d has no left child\n", parentData);
    }
    if (parent->right != nullptr) {
        printf("right child of %d: %d\n", parentData, parent->right->data);
    } else {
        printf("%d has no right child\n", parentData);
    }
    return parent;
}
```

Listing 12.10: 3 has 1 and 5; 7 has only 8; 9 is a leaf; 4 is absent. Trusting `search` crashes on 4 at address `0x8`: the field `left` of a null pointer, 8 bytes in. Book: Section 12.12.

Problem 12.10 — A Tree Becomes a List

> **PROBLEM.** Write a recursive function to convert the binary tree to a doubly linked list in place. The left pointer acts as the previous pointer and the right pointer as the next pointer. `Node *convertToDLL(Node *root)`

An **inorder** walk visits the nodes in list order; append each visited node after `prev` with two arrows: `prev->right = root`, `root->left = prev`.

`head` and `prev` must be *shared* by all calls, so the helper receives their addresses, `Node **`.

Overwriting is safe: `root->left` changes only after the left subtree is finished, and `root->right` was already read and passed down. Book: Sections 12.13.1–12.13.3.

The Same Eight Nodes, Rewired

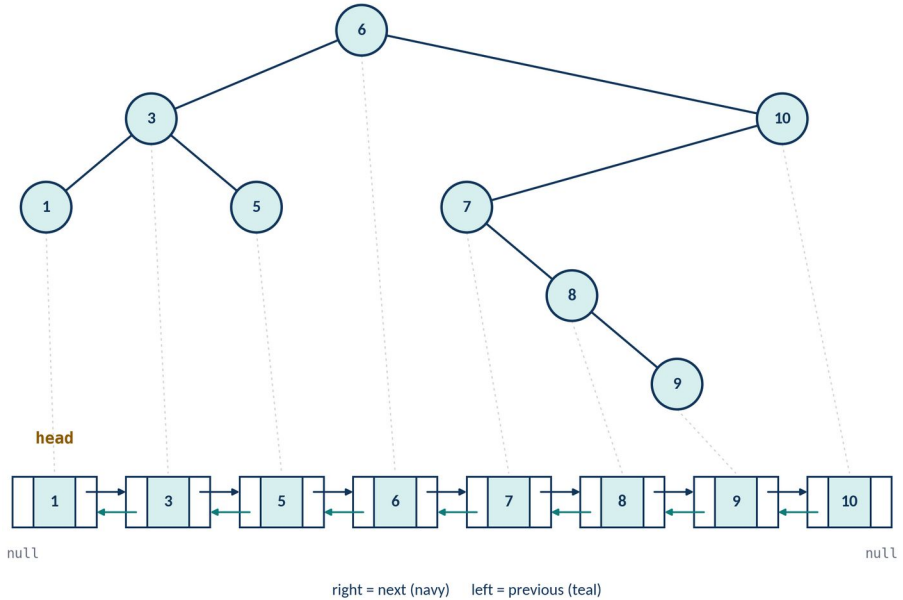


Figure 12.10 Problem 12.10: the same eight nodes, rewired. In the list, `right` points to the next larger value and `left` to the next smaller one; the dashed guides show that the list order is the inorder of the tree.

Figure 12.10: `right` now points to the next larger value, `left` to the next smaller. Book: Section 12.13.2 and Figure 12.10.

The Inorder Append

Listing 12.11 The helper and `convertToDLL` from `p1210_to_dll.cpp`.

```
void convertToDLLHelper(Node *root, Node **head, Node **prev) {
    if (root == nullptr) return;
    convertToDLLHelper(root->left, head, prev);   // 1. every smaller value first
    if (*prev == nullptr) {
        *head = root;  // 2. nothing before: root is the head
    } else {
        (*prev)->right = root;                    //    link   prev -> root
    }
    root->left = *prev;                           //    and    prev <- root
    *prev = root;                                 //    root is now the last node
    convertToDLLHelper(root->right, head, prev);  // 3. every larger value after
}

Node *convertToDLL(Node *root) {
    Node *head = nullptr;
    Node *prev = nullptr;                         // the last node linked so far
    convertToDLLHelper(root, &head, &prev);
    return head;
}
```

Listing 12.11: forward `1 3 5 6 7 8 9 10`, backward `10 9 8 7 6 5 3 1`. The nodes are now a list: free them with a walk along `right`, never with the tree's `clear`. Java and Python keep `head` and `prev` in an object. Book: Sections 12.13.4–12.13.5.

Wrong: `prev` Passed by Value

```
void convertToDLLHelper(Node *root, Node **head, Node *prev) {  // WRONG: a copy
    ...
    prev = root;                                    // changes the copy only
    convertToDLLHelper(root->right, head, prev);
}
```

```
forward : 6 10
backward: 10 6
SUMMARY: AddressSanitizer: 144 byte(s) leaked in 6 allocation(s).
```

A copy carries information down, never back up: 3 and 6 each believe they are first and cut off everything before them. A variable the calls must share is passed by pointer or held in a shared object. Book: Section 12.13.6.

Problem 12.11 — Reversing a Doubly Linked List

PROBLEM. Reverse a member's reading history, a doubly linked list with `head` and `tail`, in place, without creating any node or moving any book identifier.

Listing 12.12 `reverse` from `p1211_reverse_dll.cpp`: swap the two arrows of every node, then swap the ends.

```
void reverse(DList &list) {
    DNode *cur = list.head;
    while (cur != nullptr) {
        DNode *saved = cur->next;          // 1. remember where to go next
        cur->next = cur->prev;             // 2. swap the two arrows
        cur->prev = saved;
        cur = saved;                       // 3. move on to the old "next"
    }
    DNode *oldHead = list.head;            // 4. the two ends swap roles
    list.head = list.tail;
    list.tail = oldHead;
}
```

Listing 12.12: move to the saved old `next`. The wrong `cur = cur->next` follows the arrow just swapped, stops after one node, and leaks three. $\Theta(n)$ time, $O(1)$ memory. Book: Section 12.14.1.

Only the Arrows Change

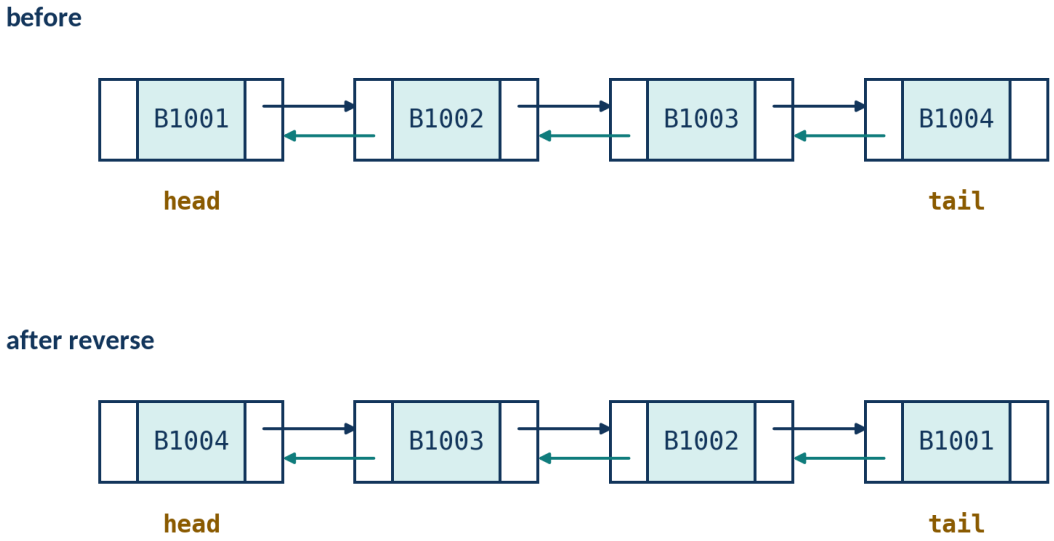


Figure 12.11 Problem 12.11: every node stays where it is in memory and keeps its book; only its two arrows exchange roles, and `head` and `tail` swap.

Figure 12.11: every node stays in place with its book; reversing twice gives the original back. Book: Section 12.14.1 and Figure 12.11.

Problem 12.12 — The Josephus Circle

n members in a circle; every k-th member counted leaves; the last one keeps the advance copy. A circular list models it: counting around is following `next` forever.

$$J(1)=0,\ J(n)=\big(J(n-1)+k\big)\bmod n\ (n\geq 2). \tag{12.8}$$

For $n=7$, $k=3$: $J=0,1,1,0,3,0,3$; position 3 is member **4**.

Equation (12.8): after the first member leaves, the smaller circle is the same problem renumbered from position k. It turns an $O(nk)$ simulation into $O(n)$ arithmetic. Book: Section 12.14.2 and Equation (12.8).

Worked Example: $n=7$, $k=3$

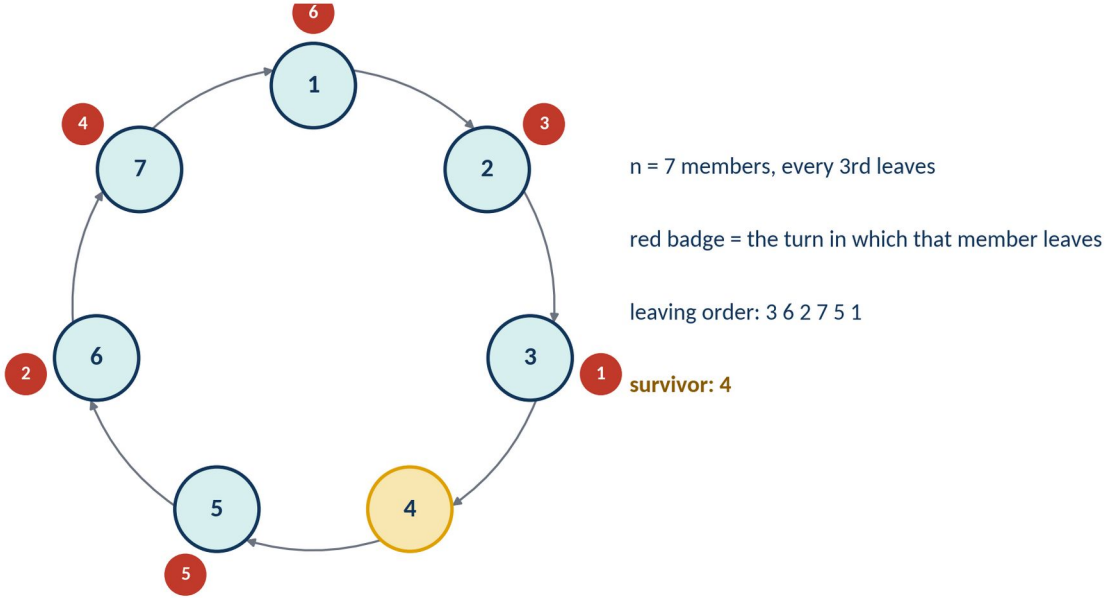


Figure 12.12 The Josephus circle for $n=7$, $k=3$: the red badge beside each member is the round in which that member leaves. Member 4 is never counted third and keeps the book.

Figure 12.12: members leave in the order 3, 6, 2, 7, 5, 1; member 4 is never counted third. Book: Section 12.14.2 and Figure 12.12.

The Circle and the Formula

Listing 12.13 `josephusList` from `p1212_josephus.cpp`; `josephusFormula` beside it computes Equation (12.8).

```
int josephusList(int n, int k) {
    CNode *prev = buildCircle(n);  // the node BEFORE the next one counted
    printf("n=%d, k=%d, leaving in order:", n, k);
    while (prev->next != prev) {                // more than one member remains
        for (int i = 1; i < k; i++) prev = prev->next;   // count k-1 members
        CNode *out = prev->next;                // the k-th member leaves
        prev->next = out->next;                 // link around it
        printf(" %d", out->member);
        delete out;
    }
    int survivor = prev->member;
    delete prev;
    printf(" | survivor %d\n", survivor);
    return survivor;
}
```

Listing 12.13: `prev` *starts at the last node, the one before member 1. Starting at member 1 instead makes every count one late, and only the independent formula exposes it. Book: Section 12.14.2.*

Problem 12.13 — Choosing Books

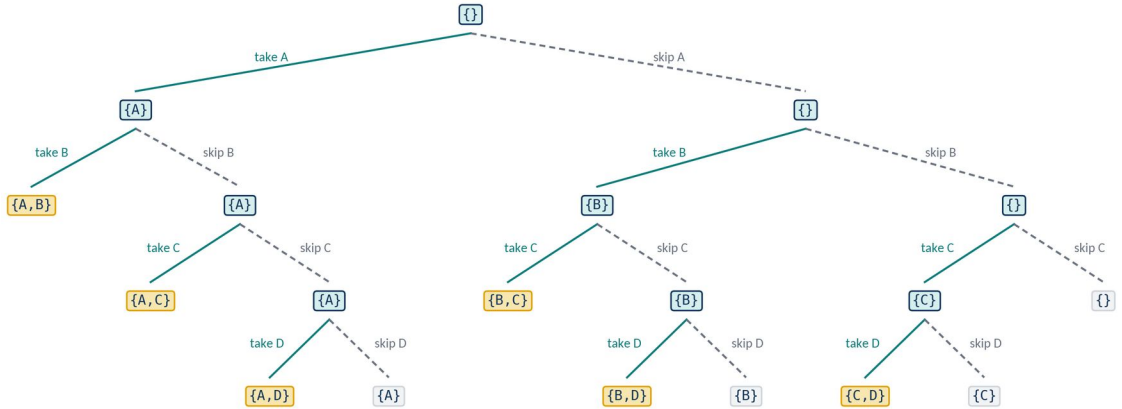
Print every choice of three of the five books `B1001` to `B1005`. Each choice either **takes** the next book and needs $k-1$ more, or **skips** it and needs k more:

$$\binom{n}{k}=\binom{n-1}{k-1}+\binom{n-1}{k},\quad \binom{n}{0}=\binom{n}{n}=1. \tag{12.9}$$

Base cases: $k=0$, print the choice; fewer than k books left, give up (*pruning*).

Equation (12.9) is Pascal's triangle. The recursive `binom(20, 10)` makes 369,511 calls, repeating subproblems like naive Fibonacci; a table would cut that to about $n\,k$. Book: Section 12.15.1 and Equation (12.9).

Take or Skip



amber: a complete choice (k = 0), printed; grey: too few books left, abandoned

Figure 12.13 The recursion of `choose` for two books out of A, B, C, D. Each call either takes or skips the next book; the six amber leaves are the six choices, and the grey leaves are branches abandoned because too few books remained.

Figure 12.13: two of A, B, C, D: sixteen calls, six amber choices, three dead ends. Book: Section 12.15.1 and Figure 12.13.

Backtracking in C++

Listing 12.14 `choose` from `p1213_choose_books.cpp`: append, explore, remove.

```
void choose(const vector<string> &books, int start, int k,
            vector<string> &picked, int &ways) {
    if (k == 0) {                                   // a complete choice: print it
        ways++;
        printf("%2d:", ways);
        for (const string &b : picked) printf(" %s", b.c_str());
        printf("\n");
        return;
    }
    int left = (int)books.size() - start;           // books not yet considered
    if (left < k) return;                           // too few left: a dead end
    picked.push_back(books[start]);                 // branch 1: take books[start]
    choose(books, start + 1, k - 1, picked, ways);
    picked.pop_back();                              // undo that choice
    choose(books, start + 1, k, picked, ways);      // branch 2: skip books[start]
}
```

Listing 12.14: ten choices in catalog order, and $\binom{5}{3}=10$. *Forget* `pop_back` *and each line grows longer. Whatever a call receives by value is undone for free; shared state it must undo itself. Book: Section 12.15.1.*

Problem 12.14 — Is It Really a BST?

Listing 12.15 `inRange` and `isBST` from `p1214_is_bst.cpp`: every node checked against the interval inherited from its ancestors.

```
bool inRange(Node *root, long long low, long long high) {
    if (root == nullptr) return true;
    if (root->data <= low || root->data >= high) return false;
    return inRange(root->left, low, root->data) &&     // left: below this value
           inRange(root->right, root->data, high);     // right: above this value
}

bool isBST(Node *root) {
    return inRange(root, LLONG_MIN, LLONG_MAX);        // no limits at the root
}
```

Listing 12.15: going left the value becomes `high`, going right `low`; open intervals reject equal values. Comparing each node only with its children wrongly accepts the sneaky tree. Book: Section 12.16.1.

The Sneaky Tree

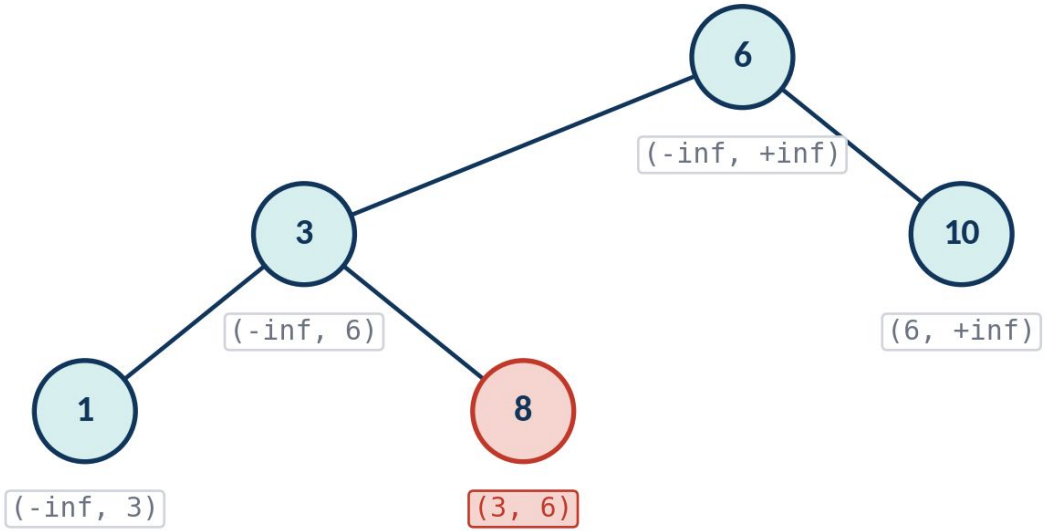


Figure 12.14 The interval each call of `inRange` receives. Every parent–child pair is in order, but the value 8 lies outside its interval $(3, 6)$, so the tree is not a binary search tree.

Figure 12.14: every parent-child pair is in order, but 8 lies outside $(3, 6)$; `search(8)` would miss it. Its inorder `1 3 8 6 10` is not increasing. Book: Section 12.16.1 and Figure 12.14.

Problem 12.15 — Lowest Common Ancestor

Listing 12.16 `lca` from `p1215_lca.cpp`: walk down while both values lie on the same side.

```
Node *lca(Node *root, int a, int b) {
    if (root == nullptr) return nullptr;
    if (a < root->data && b < root->data) {     // both on the left
        return lca(root->left, a, b);
    }
    if (a > root->data && b > root->data) {     // both on the right
        return lca(root->right, a, b);
    }
    return root;                // they part here, or one of them is here
}
```

Listing 12.16: lca(1, 5) = 3, lca(5, 9) = 6, lca(8, 9) = 8, lca(9, 9) = 9. Steering by one value alone just walks to a. One path, $O(h)$. Book: Section 12.16.2.

Where the Paths Part

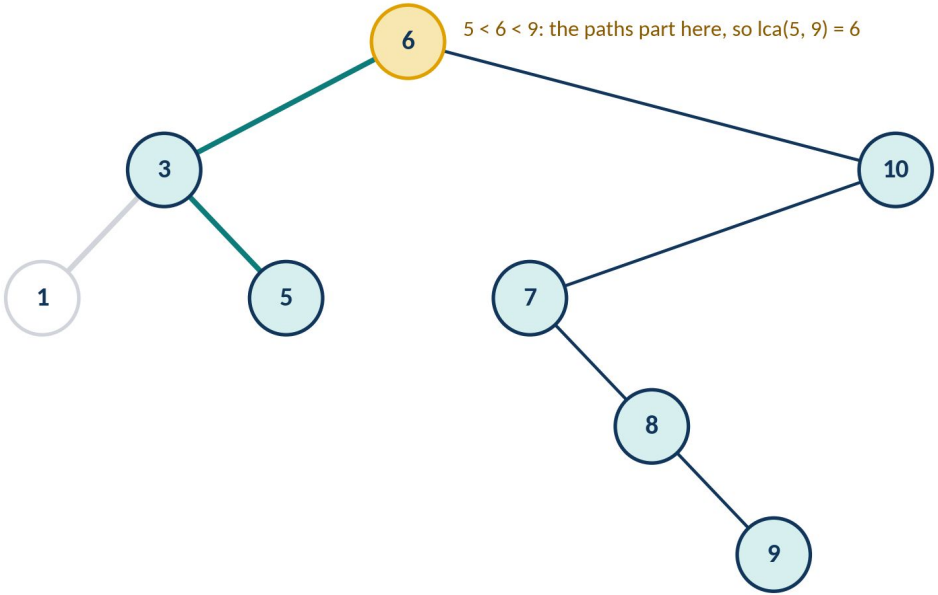


Figure 12.15 The paths from the root to 5 (teal) and to 9 (navy) share only the root: since $5<6<9$, the two values part at 6, which is their lowest common ancestor.

Figure 12.15: since $5<6<9$, the paths part at the root. Book: Section 12.16.2 and Figure 12.15.

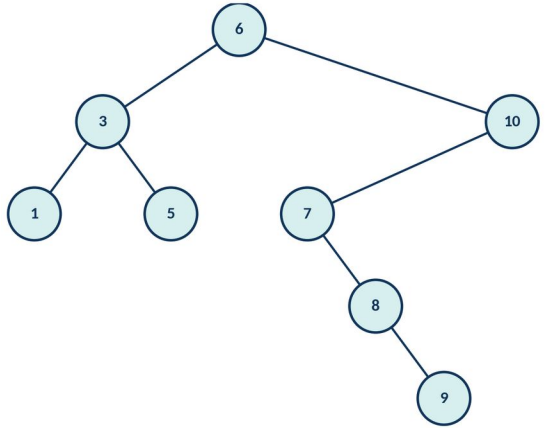
Problem 12.16 — Level by Level

Listing 12.17 `printLevels` from `p1216_levels.cpp`: the queue's size, read once, is the width of the level.

```
int printLevels(Node *root) {  // returns the width of the widest level
    int widest = 0;
    if (root == nullptr) return widest;
    std::queue<Node *> q;
    q.push(root);
    for (int level = 0; !q.empty(); level++) {
        int width = (int)q.size();            // exactly the nodes of this level
        if (width > widest) widest = width;
        printf("level %d:", level);
        for (int i = 0; i < width; i++) {
            Node *cur = q.front();
            q.pop();
            printf(" %d", cur->data);
            if (cur->left != nullptr) q.push(cur->left);     // they belong to
            if (cur->right != nullptr) q.push(cur->right);   // the next level
        }
        printf("\n");
    }
    return widest;
}
```

Listing 12.17: $\Theta(n)$ *instead of Problem 12.2's* $\Theta(n^2)$. *Reading* `q.size()` *inside the loop condition lets levels run together:* `level 0: 6 3 10`. *Book: Section 12.16.3.*

The Queue, One Level at a Time



level	width	dequeued (printed)	enqueued
0	1	6	3 10
1	2	3 10	1 5 7
2	3	1 5 7	8
3	1	8	9
4	1	9	-

Figure 12.16 Problem 12.16: at the start of each level the queue holds exactly that level; its size is the level's width. The nodes taken from the queue are printed, and their children are enqueued for the next level.

Figure 12.16: widths 1, 2, 3, 1, 1; a perfect tree of height 3 gives 1, 2, 4, 8. Book: Section 12.16.3 and Figure 12.16.

The Problems Compared

Every node once is $\Theta(n)$ by Equation (12.1): Problems 12.1, 12.3–12.5, 12.10, 12.14, 12.16.

One path from the root is $O(h)$: Problems 12.7–12.9 and 12.15, which is why balanced trees matter.

A walk inside a walk is the pattern to watch: the helper balance check, all levels by Problem 12.2. Let the outer walk *return* what the inner one recomputes.

Several wrong solutions *pass the example*: the balance check, the second sum, the Josephus order. Steps 6 and 7 catch them.

The book's comparison table lists, for all sixteen problems, the structure, the cost and the input on which the wrong solution fails. Book: Section 12.17.

Profitina in Practice

A threaded discussion is a tree: reading order is Problem 12.1, indentation is the level of Problem 12.2, reply counts are the sum of Problem 12.3; deleting a post with replies raises Problem 12.7's question.

A site is walked level by level: a scanner with a fixed depth uses a queue, as in Problem 12.16.

Recursion has a depth limit: a Python renderer nested a thousand replies deep raises `RecursionError`; use an explicit stack or cap the nesting.

Profitina is an AI-visibility (GEO) platform; we describe the shapes of these problems in such a product, not its implementation. Book: Section 12.18.

Common Mistakes

No base case, or the wrong one: 0 for `+`, true for `&&`, `INT_MIN` for max, -1 for height. **Testing `k == 0` before `nullptr`.**

Communicating through a global; passing a node by value; swapping without saving.

Throwing away the returned subtree; using BST functions on a tree that is not a BST; checking only parent and child.

Following a pointer just overwritten; forgetting to undo in backtracking; recursing too deep; testing only the example.

Each mistake is the tempting wrong solution of one of today's problems. Book: Section 12.19.

Summary: Five Things to Remember

1. Seven steps: example, empty answer, arguments drawn, root-left-right, re-attach, edge cases, tests under a memory checker.

2. The base case is the value that changes nothing under the combining step.

3. A function that may replace its node returns the replacement, and every caller re-attaches it.

4. Share state only on purpose (by pointer or object), save before you overwrite, undo what you changed.

5. Every node once is $\Theta(n)$, one path is $O(h)$; beware a walk inside a walk.

Book: Section 12.20.

Next Time

Chapter 13 — Graphs. Trees whose links may form cycles: vertices, edges, adjacency, breadth-first and depth-first search.

The queue of Problem 12.16 and the stack of Problem 12.1 return as BFS and DFS.

Before then: work Exercises 12.1, 12.4, 12.7 and 12.12, in C++ and then in Java or Python.

The solutions, with programs, are in the book. Book: Sections 12.21 and 12.22.

Lecture 13 — Graphs

Course: Data Structure (EF234201)

Book: Irfan Subakti, *Data Structure with C++, Java and Python*, Chapter 13

Lecturer: Irfan Subakti — yifana@gmail.com

Today the last restriction goes: things and connections between pairs of things, with any number of connections and loops allowed. A list is a graph, a tree is a graph, and so is almost everything else. Book: Chapter 13, Sections 13.1–13.15.

What This Meeting Covers

1. **Vocabulary** — vertices, edges, directed and weighted graphs, degree and the handshake lemma, paths, cycles, components, DAGs.
2. **Representations** — adjacency matrix, adjacency list, edge list, and their costs.
3. **Breadth-first and depth-first search** — the queue and the stack at work, with proofs and traces.
4. **Three applications** — connected components, topological sort, and Dijkstra’s shortest paths.
5. **At scale** — road networks and social graphs; a website audit for Profitina Laundry.

The running example is the Pustaka Ceria campus map: ten buildings, eleven footpaths. Book: Section 13.1.

Where We Left Off

Arrays and lists put elements in a **line**; stacks and queues restrict where we touch it; a tree allows many children but **one parent**.

Campus footpaths, friendships, flights, web links and task dependencies obey no such rule, and they form loops.

The **queue** of Chapter 5 drives breadth-first search; the **stack**, or recursion, drives depth-first search; the **priority queue** of Chapter 14 drives Dijkstra.

Each section answers one practical question: can I get there, by how few hops, how many meters, what is cut off, in what order must tasks run? Book: Section 13.1.

The Bridges of Königsberg

Seven bridges joined four pieces of land. Is there a walk that crosses every bridge exactly once?

Euler, 1736, kept only **which** land masses exist and **which pairs** are joined, and how often.

A walk passing *through* a land mass uses its bridges two at a time, so all but the start and the end need an even count.

Here all four are odd (3, 5, 3, 3): impossible.

Points and connections, a count per point (the degree), and reasoning about walks: that is graph theory in miniature, 290 years old. Book: Section 13.2.1.

Königsberg as a Graph

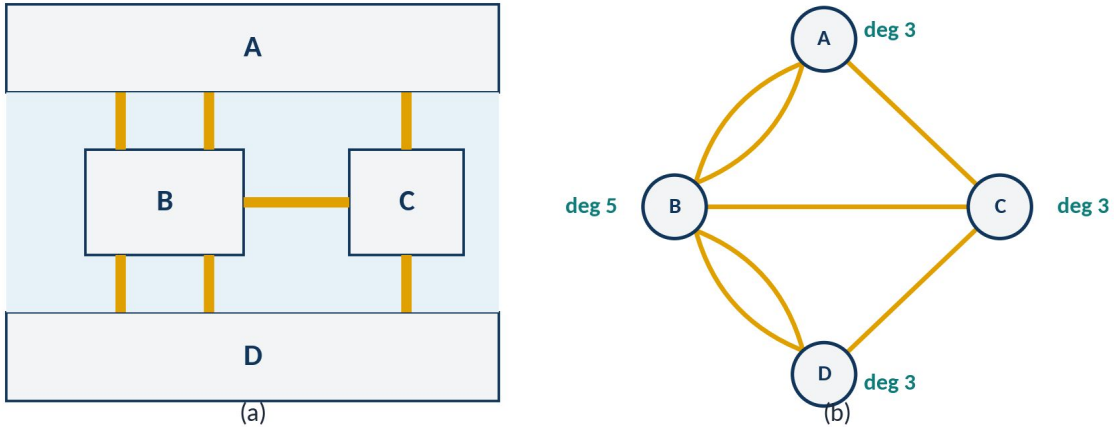


Figure 13.1 The seven bridges of Königsberg: (a) a schematic map with the four land masses A to D and the seven bridges in amber; (b) the same situation as a graph, with the number of bridges touching each land mass.

Figure 13.1: the argument is a proof, not a failure to search hard enough. Book: Section 13.2.1 and Figure 13.1.

Vertices and Edges

DEFINITION. A *graph* G consists of a finite set V of *vertices* and a set E of *edges*, where each edge joins a pair of vertices, as in Equation (13.1).

$$G=(V,E),E\subseteq\{\{u,v\}:u,v\in V,u\neq v\} \tag{13.1}$$

Joined vertices are *adjacent* (neighbors). $u\neq v$ forbids self-loops; E being a set forbids parallel edges: the graph is *simple*.

Königsberg, with two bridges between A and B, is a *multigraph*. We write V and E for $|V|$ and $|E|$ in formulas of running time.

This chapter says vertex and edge throughout and keeps node for linked lists. Book: Section 13.3.1 and Equation (13.1).

Undirected, Directed, Weighted

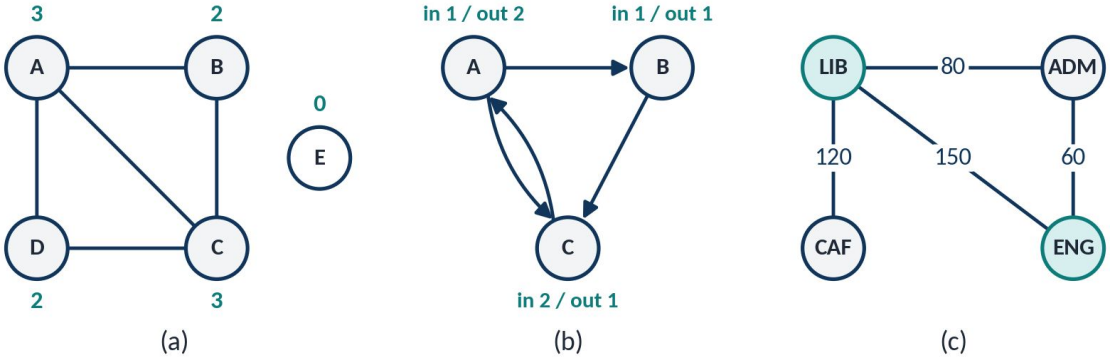


Figure 13.2 Three small graphs: (a) an undirected graph with the degree of each vertex written beside it, including the isolated vertex E; (b) a directed graph with the in-degree and out-degree of each vertex; (c) a weighted graph, a corner of the campus map with footpath lengths in meters.

Figure 13.2: (a) E is an isolated vertex; (b) arrows are ordered pairs (u, v); (c) $w(\text{LIB}, \text{ADM}) = 80$ meters. An undirected edge is stored as two arrows. Book: Sections 13.3.1–13.3.3 and Figure 13.2.

Degree and the Handshake Lemma

$$\sum_{v \in V} \deg(v) = 2|E| \tag{13.2}$$

Proof. Count (vertex, incident edge) pairs two ways: by vertex, $\sum \deg(v)$; by edge, two per edge.

So the number of odd-degree vertices is even: degrees 3, 3, 3, 3, 3 are impossible. In a directed graph:

$$\sum_{v \in V} \deg^-(v) = \sum_{v \in V} \deg^+(v) = |E| \tag{13.3}$$

Equation (13.2) is a free test for a program that builds a graph; Equation (13.3) counts in-degrees, the basis of Kahn's algorithm. In Figure 13.2(a): $3+2+3+2+0 = 10 = 2 \cdot 5$. *Book: Section 13.3.4.*

Paths, Cycles, Components, DAGs

A **path** is a sequence of vertices joined by edges; its length is the number of edges; v is **reachable** from u if a path exists.

A **cycle** returns to its start. Trees are exactly the connected acyclic undirected graphs, so graph algorithms must remember what they have seen.

> **DEFINITION.** A *connected component* of an undirected graph is a maximal set of vertices in which every vertex is reachable from every other.

A **DAG** is a directed graph without a directed cycle: dependencies.

"Is reachable from" is an equivalence relation on an undirected graph, and the components are its classes. Book: Sections 13.3.5–13.3.7.

Sparse and Dense

$$|E| \leq \binom{|V|}{2} = \frac{|V|(|V|-1)}{2} \tag{13.4}$$

$$\rho = \frac{|E|}{|V|(|V|-1)/2} \tag{13.5}$$

Dense: close to Equation (13.4); sparse: a small constant times $|V|$. Equation (13.5) is 1 for the complete graph.

In a road network almost every junction has at most four or five roads, so $|E|$ grows like $|V|$: the graphs that matter most are overwhelmingly sparse. Book: Section 13.3.8.

The Pustaka Ceria Campus Map

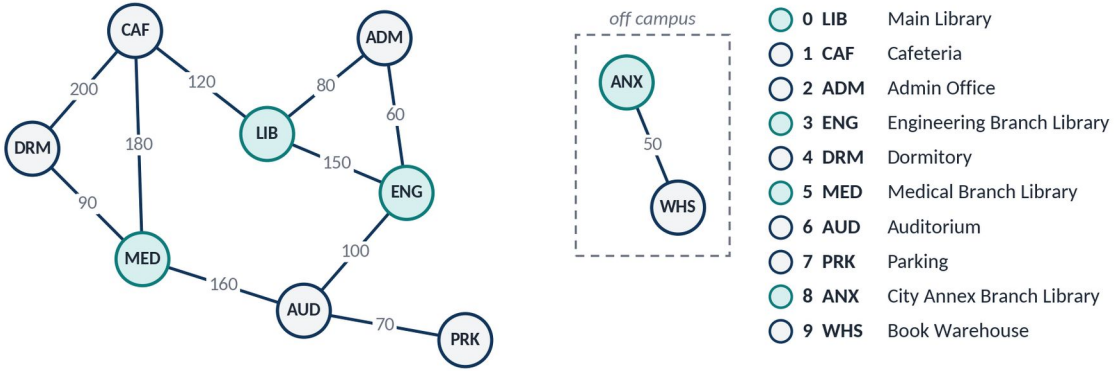


Figure 13.3 The Pustaka Ceria campus map, the running example of this chapter: ten buildings (libraries in teal) and eleven footpaths with their lengths in meters. The city annex and the warehouse are off campus and not yet connected to it.

Figure 13.3: undirected and weighted, full of cycles, and disconnected (ANX and WHS). The running example from here on. Book: Section 13.4 and Figure 13.3.

Worked Example: the Map by the Numbers

$|V| = 10$, $|E| = 11$. Degrees 3, 3, 2, 3, 2, 3, 3, 1, 1, 1: sum $22 = 2 \times 11$.

Eight vertices of odd degree: an even number, as the lemma demands.

At most $10 \cdot 9/2 = 45$ edges, so the density is $11/45 \approx 0.24$.

Hops and meters disagree: LIB, ENG, AUD, PRK is 3 footpaths and 320 m; LIB, ADM, ENG, AUD, PRK is 4 footpaths and 310 m.

BFS will find the first route, Dijkstra the second. Book: Section 13.4.

Three Representations

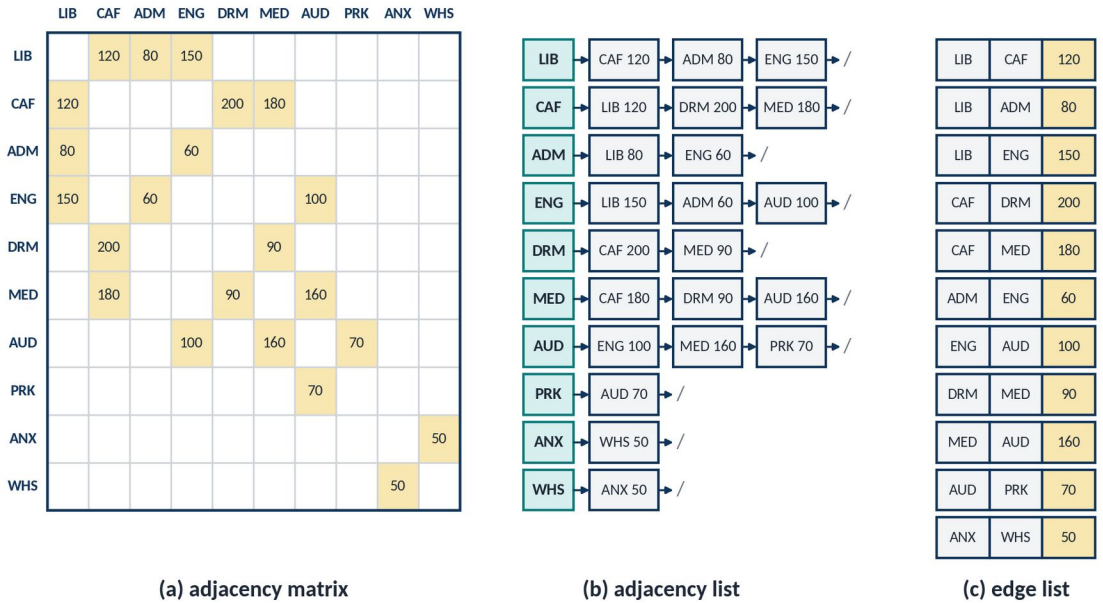


Figure 13.4 The campus map stored three ways: (a) the adjacency matrix, with footpath lengths in the shaded cells; (b) the adjacency list, one chain of (neighbor, length) entries per vertex; (c) the edge list, one record per footpath.

Figure 13.4: the matrix is symmetric with an empty diagonal and 22 of 100 cells used; each list holds (neighbor, length) entries; the edge list has one record per footpath. Book: Section 13.5 and Figure 13.4.

Matrix or List?

Matrix: “is there an edge?” in $O(1)$; listing neighbors scans a whole row, $\Theta(V)$.

List: neighbors in $\Theta(\deg(u))$; an edge test scans `adj[u]`. An undirected edge appears twice.

$$S_{\text{matrix}} = |V|^2 \text{ cells}, S_{\text{list}} = |V| + 2|E| \text{ entries} \tag{13.6}$$

> **RULE OF THUMB.** Use an adjacency list unless the graph is dense or the algorithm mostly asks “is there an edge?”. Keep an edge list when edges arrive and are processed one by one.

Equation (13.6): 100 cells against 32 entries on the campus map. In Python, `[[0] * n] * n` makes one row shared `n` times: use `[[0] * n for _ in range(n)]`. Book: Sections 13.5.1–13.5.6 and Equation (13.6).

The Costs Side by Side

Table 13.1 The costs of the three graph representations, for a graph with V vertices and E edges.

Operation	Adjacency matrix	Adjacency list	Edge list
Memory	$\Theta(V^2)$	$\Theta(V+E)$	$\Theta(E)$
Is $\{u, v\}$ an edge?	$O(1)$	$O(\deg(u))$	$O(E)$
List the neighbors of u	$\Theta(V)$	$\Theta(\deg(u))$	$\Theta(E)$
Degree of u	$\Theta(V)$	$O(1)$	$\Theta(E)$
Add an edge	$O(1)$	$O(1)$ amortized	$O(1)$ amortized
Remove an edge	$O(1)$	$O(\deg(u)+\deg(v))$	$O(E)$
Add a vertex	$\Theta(V^2)$ (copy the matrix)	$O(1)$ amortized	$O(1)$
Visit every edge once	$\Theta(V^2)$	$\Theta(V+E)$	$\Theta(E)$
Best for	dense graphs, many edge queries	sparse graphs, traversals	input files, edge-by-edge algorithms

Table 13.1: the deciding row is "visit every edge once": $\Theta(V+E)$ with lists against $\Theta(V^2)$ with a matrix. Book: Section 13.5.4.

Breadth-First Search: Ripples in a Pond

From a source s: first s, then every vertex one edge away, then two, then three: rings of equal **hop distance**.

The **FIFO queue** keeps the rings in order: ring 1 enters before any vertex of ring 2 is discovered.

For every vertex BFS keeps $d[v]$ (hop distance, -1 = undiscovered) and $\pi[v]$ (the parent that discovered it).

Moore (1959) used it for mazes, Lee (1961) for routing wires on circuit boards. Book: Section 13.6.1.

The Procedure

Pseudocode 13.1 BFS: breadth-first search from the source s, computing hop distances d and parents π.

```
BFS(G, s):
 1.  for each vertex v in G.V
 2.      v.d ← -1                    // -1 means "not discovered yet"
 3.      v.π ← NIL
 4.  s.d ← 0
 5.  Q ← an empty queue
 6.  ENQUEUE(Q, s)
 7.  while Q is not empty
 8.      u ← DEQUEUE(Q)
 9.      for each v in G.adj[u]
10.          if v.d = -1             // first time v is seen
11.              v.d ← u.d + 1
12.              v.π ← u
13.              ENQUEUE(Q, v)
```

*Pseudocode 13.1: a vertex is **marked when it enters** the queue, so it enters at most once; d doubles as the visited mark; the parents form the BFS tree. Book: Section 13.6.2.*

Worked Example: BFS from LIB

Table 13.2 Breadth-first search from LIB on the campus map, step by step.

Step	Dequeued u	$d[u]$	Neighbors of u in list order	Newly discovered	Queue afterward (front first)
0				LIB ($d=0$)	LIB
1	LIB	0	CAF, ADM, ENG	CAF, ADM, ENG ($d=1$, parent LIB)	CAF ADM ENG
2	CAF	1	LIB, DRM, MED	DRM, MED ($d=2$, parent CAF)	ADM ENG DRM MED
3	ADM	1	LIB, ENG	none (both seen)	ENG DRM MED
4	ENG	1	LIB, ADM, AUD	AUD ($d=2$, parent ENG)	DRM MED AUD
5	DRM	2	CAF, MED	none	MED AUD
6	MED	2	CAF, DRM, AUD	none	AUD
7	AUD	2	ENG, MED, PRK	PRK ($d=3$, parent AUD)	PRK
8	PRK	3	AUD	none	(empty)

Table 13.2: at step 3 ENG is refused although it is still in the queue; at step 6 MED leaves AUD alone. ANX and WHS never enter. Book: Section 13.6.3.

The Queue After Each Step

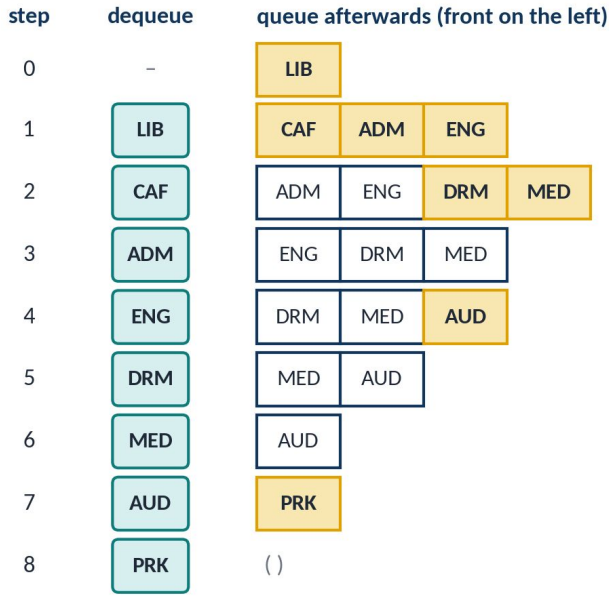


Figure 13.5 The BFS queue after each step of Table 13.2: the dequeued vertex on the left, the queue contents on the right with newly enqueued vertices in amber.

Figure 13.5: the queue always holds one ring, or the end of one ring followed by the start of the next. Book: Section 13.6.3 and Figure 13.5.

The Map, Ring by Ring

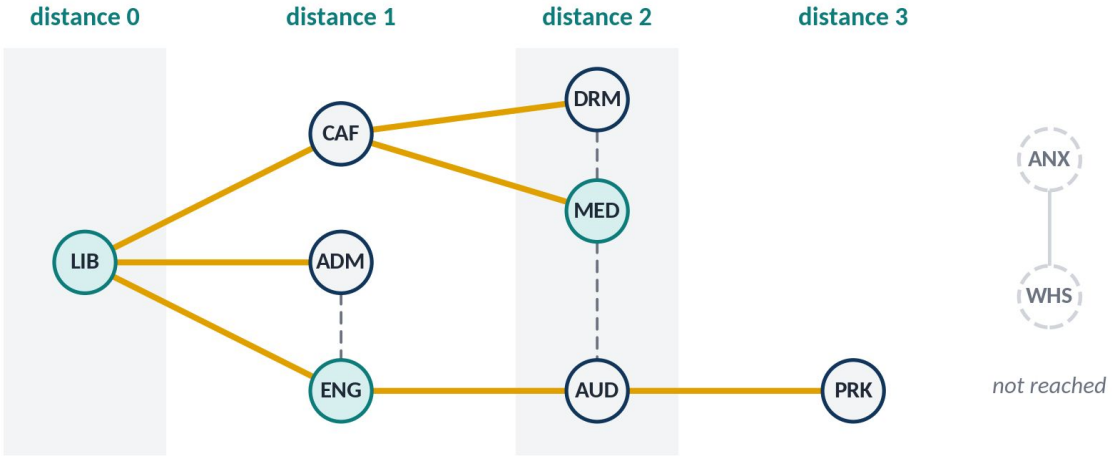


Figure 13.6 The campus map arranged by hop distance from LIB. Amber edges form the BFS tree; dashed edges were examined but led to vertices already discovered; ANX and WHS are not reached.

Figure 13.6: amber edges form the BFS tree; the dashed edges join vertices of the same ring or of neighboring rings. No edge skips a ring. Book: Section 13.6.3 and Figure 13.6.

Why BFS Finds Fewest-Hop Routes

Step 1: $d[v]$ is never too small: the parent chain is a real path of $d[v]$ edges.

Step 2: the queue is sorted by d, holding at most two neighboring values, so vertices leave in nondecreasing d.

Step 3: $d[v]$ is never too large: a smallest counterexample contradicts Step 2. Hence for every edge, Equation (13.7):

$$d[v] \le d[u] + 1 \tag{13.7}$$

$$T_{\text{BFS}} = \Theta(V) + \sum_{u \in V} O(1 + \deg(u)) = O(V + E) \tag{13.8}$$

Equation (13.8): every vertex enters and leaves once and every list is scanned once: linear time for graphs. With a matrix it would be $\Theta(V^2)$. Book: Sections 13.6.4–13.6.5.

BFS in C++

Listing 13.1 The heart of `bfs` in `bfs.cpp`: `dist` doubles as the visited mark, set when a vertex enters the queue.

```
    dist.assign(V, -1);
    parent.assign(V, -1);
    std::vector<int> order;
    std::queue<int> q;
    dist[s] = 0;                 // mark s when it ENTERS the queue
    q.push(s);
    int step = 0;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        order.push_back(u);
        std::string added;
        for (int v : adj[u]) {
            if (dist[v] == -1) { // first time we see v
                dist[v] = dist[u] + 1;
                parent[v] = u;
                q.push(v);
                added += std::string(" ") + CODE[v];
            }
        }
```

Listing 13.1: the fewest-hop route to PRK is LIB, ENG, AUD, PRK; WHS keeps -1 and the program says so. Java uses `ArrayDeque`, Python `collections.deque`; a Python `list.pop(0)` would make BFS quadratic. Book: Sections 13.6.6–13.6.7.

Depth-First Search: a Ball of Thread

Theseus in the Labyrinth: take an untried corridor and go deeper; when every corridor leads to a room already seen, follow the thread back one room.

The thread is a **stack**; recursion keeps it for us.

Colors: **white** undiscovered, **gray** on the thread, **black** finished. Two time stamps: discovery $v.d$ and finish $v.f$.

BFS explores cautiously, ring by ring; DFS boldly, as far as it can along one route. The modern form is Tarjan's (1972). Book: Sections 13.7.1–13.7.2.

DFS and DFS-VISIT

Pseudocode 13.2 DFS and DFS-VISIT: depth-first search with discovery and finish times.

```
DFS(G):
 1.  for each vertex u in G.V
 2.      u.color ← WHITE
 3.      u.π ← NIL
 4.  time ← 0
 5.  for each vertex u in G.V
 6.      if u.color = WHITE
 7.          DFS-VISIT(G, u)

DFS-VISIT(G, u):
 1.  time ← time + 1
 2.  u.d ← time                    // u is discovered
 3.  u.color ← GRAY
 4.  for each v in G.adj[u]
 5.      if v.color = WHITE          // an unexplored corridor
 6.          v.π ← u
 7.          DFS-VISIT(G, v)         // go deeper
 8.  u.color ← BLACK
 9.  time ← time + 1
10.  u.f ← time                    // u is finished
```

Pseudocode 13.2: one call per vertex; $\Theta(V+E)$. Book: Section 13.7.2.

Worked Example: DFS of the Campus Map

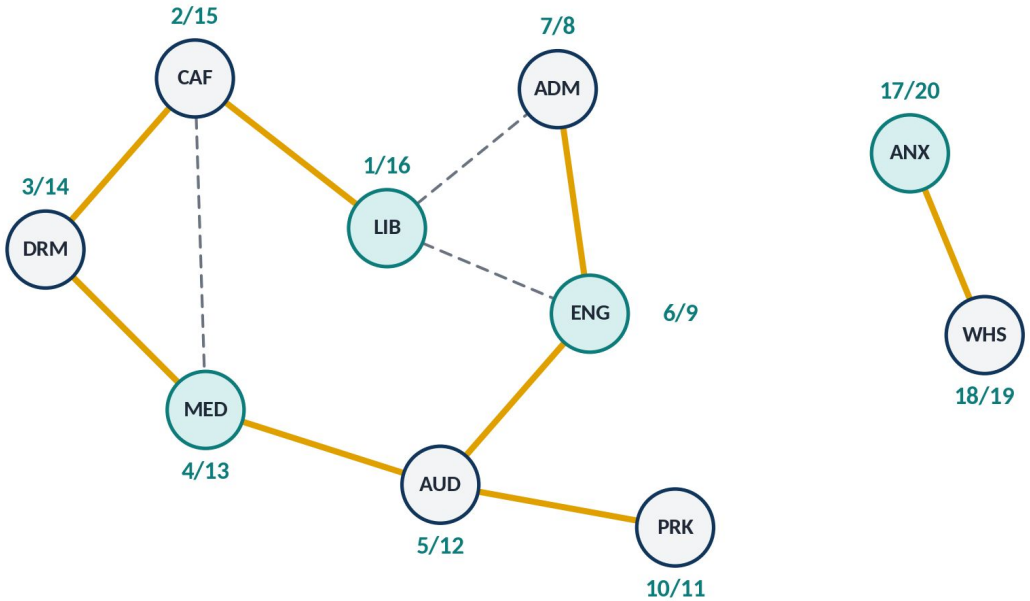


Figure 13.7 Depth-first search of the campus map. Each vertex is labeled with its discovery and finish times d/f; amber edges are tree edges and dashed edges are back edges.

Figure 13.7: LIB, CAF, DRM, MED, AUD, ENG, ADM, PRK, then ANX, WHS; six buildings deep before PRK. In an undirected graph there are only tree edges and back edges, and every back edge closes a cycle. Book: Section 13.7.3 and Figure 13.7.

The Parenthesis Theorem

$$[u.d, u.f] \cap [v.d, v.f] = \varnothing \text{ or } [u.d, u.f] \subset [v.d, v.f] \text{ or } [v.d, v.f] \subset [u.d, u.f] \tag{13.9}$$

Discovery is “(”, finish is “)”: “(LIB (CAF (DRM (MED (AUD (ENG (ADM ADM) ENG) (PRK PRK) AUD) MED) DRM) CAF) LIB) (ANX (WHS WHS) ANX)”.

A vertex discovered while u is gray is pushed on top of u and finishes first: its interval nests inside.

Equation (13.9) is the stack discipline of the thread; finish times are the key to topological sort. Book: Section 13.7.4 and Equation (13.9).

Nested or Disjoint

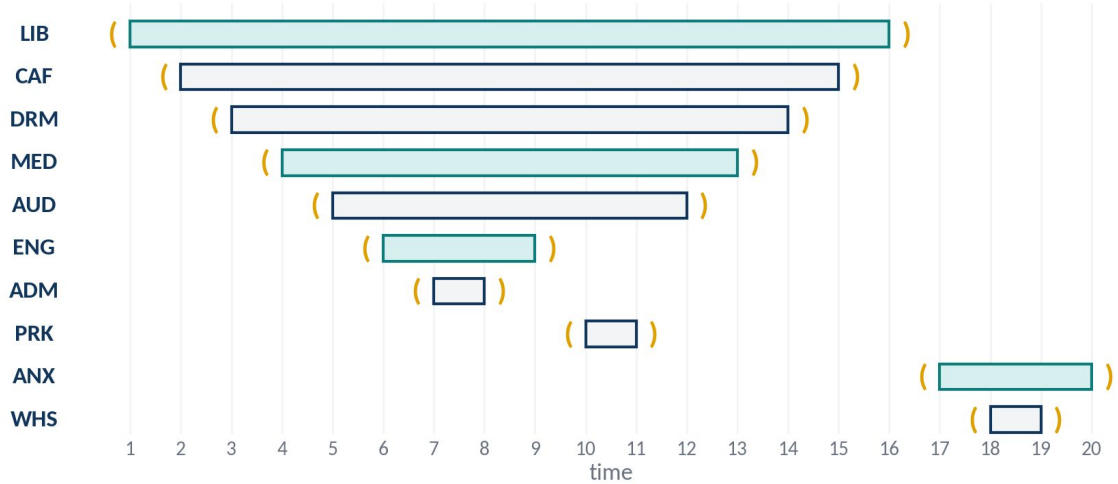


Figure 13.8 The discovery-to-finish interval of every vertex in the DFS of Figure 13.7: any two intervals are either nested or disjoint, like correctly matched parentheses.

Figure 13.8: PRK (10 to 11) lies inside AUD (5 to 12) and is disjoint from ENG (6 to 9). Book: Section 13.7.4 and Figure 13.8.

DFS Without Recursion

Pseudocode 13.3 `DFS-ITERATIVE`: the same search with an explicit stack of (vertex, next-neighbor index) pairs, giving the same discovery and finish times as the recursive version.

```
DFS-ITERATIVE(G, s):
 1.  time ← 1;  s.d ← time
 2.  S ← an empty stack;  PUSH(S, (s, 0))
 3.  while S is not empty
 4.      (u, i) ← TOP(S)
 5.      if i < length of G.adj[u]
 6.          v ← G.adj[u][i]
 7.          TOP(S).i ← i + 1         // next time, try the neighbor after v
 8.          if v.d = 0              // white: "call" DFS-VISIT(v)
 9.              time ← time + 1;  v.d ← time
10.              PUSH(S, (v, 0))
11.      else                        // no neighbor left: "return" from u
12.          time ← time + 1;  u.f ← time
13.          POP(S)
```

Pseudocode 13.3: each stack entry is (vertex, next-neighbor index), exactly what a recursive call remembers, so the time stamps are identical. The "simple" version that pushes all neighbors gives no finish times and may hold $|E|$ entries. Book: Section 13.7.5.

Recursive DFS in C++

Listing 13.2 `dfsVisit` from `dfs.cpp`: `d[v] == 0` means white; the depth only indents the printout.

```
void dfsVisit(int u, int depth) {
    d[u] = ++timer;                          // discovered
    std::printf("%*s%s d=%d\n", 2 * depth, "", CODE[u], d[u]);
    for (int v : adj[u]) {
        if (d[v] == 0) {                     // unvisited neighbor: go deeper
            parent[v] = u;
            dfsVisit(v, depth + 1);
        }
    }
    f[u] = ++timer;                          // finished: every neighbor is done
}
```

Listing 13.2. Its companion `findCycle` reports a visited neighbor that is not the vertex we came from: the cycle MED, DRM, CAF. Python stops recursion at about 1,000 levels: for large graphs, only the iterative version is safe. Book: Sections 13.7.6–13.7.7.

Connected Components

Pseudocode 13.4 `CONNECTED-COMPONENTS`: label every vertex with the number of its component, one breadth-first search per component.

```
CONNECTED-COMPONENTS(G):
 1.  for each vertex v in G.V
 2.      v.comp ← -1                 // not yet in any component
 3.  count ← 0
 4.  for each vertex s in G.V
 5.      if s.comp = -1              // s starts a new component
 6.          s.comp ← count
 7.          Q ← an empty queue;  ENQUEUE(Q, s)
 8.          while Q is not empty    // one BFS labels the whole component
 9.              u ← DEQUEUE(Q)
10.              for each v in G.adj[u]
11.                  if v.comp = -1
12.                      v.comp ← count
13.                      ENQUEUE(Q, v)
14.          count ← count + 1
15.  return count
```

Pseudocode 13.4: every vertex is labeled and enqueued once in the whole run, so the nested loops still cost $\Theta(V+E)$. Then "are u and v connected?" is `comp[u] == comp[v]`. Book: Section 13.8.1.

Worked Example: the Shuttle Road

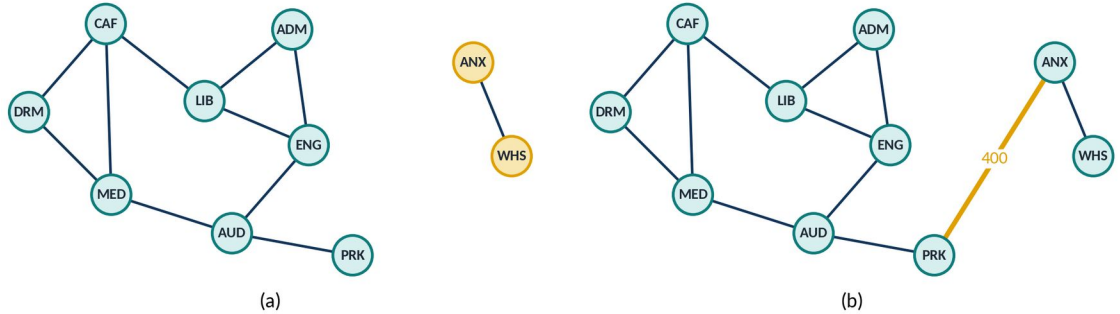


Figure 13.9 Connected components of the campus map: (a) as built, two components (teal and amber); (b) after the 400-meter shuttle road PRK-ANX is added, one component.

Figure 13.9: two components; one 400 m road PRK-ANX merges them. AUD-PRK and PRK-ANX are bridges; LIB-CAF is not. Union-find handles growing networks in almost constant time. Book: Sections 13.8.2–13.8.3 and Figure 13.9.

Topological Order

DEFINITION. A *topological order* of a directed graph lists all its vertices so that for every arrow (u, v), u comes before v.

It exists exactly when the graph is a DAG: a cycle would force a task to come before itself.

Every DAG has a vertex of in-degree 0 (walk backward along incoming arrows; a repeat would be a cycle), so Kahn's algorithm never gets stuck; on a cycle it outputs fewer than $|V|$ vertices.

Topological orders are usually not unique: inspect and catalog may be done in either order. Book: Sections 13.9.1–13.9.2.

The New-Book Workflow

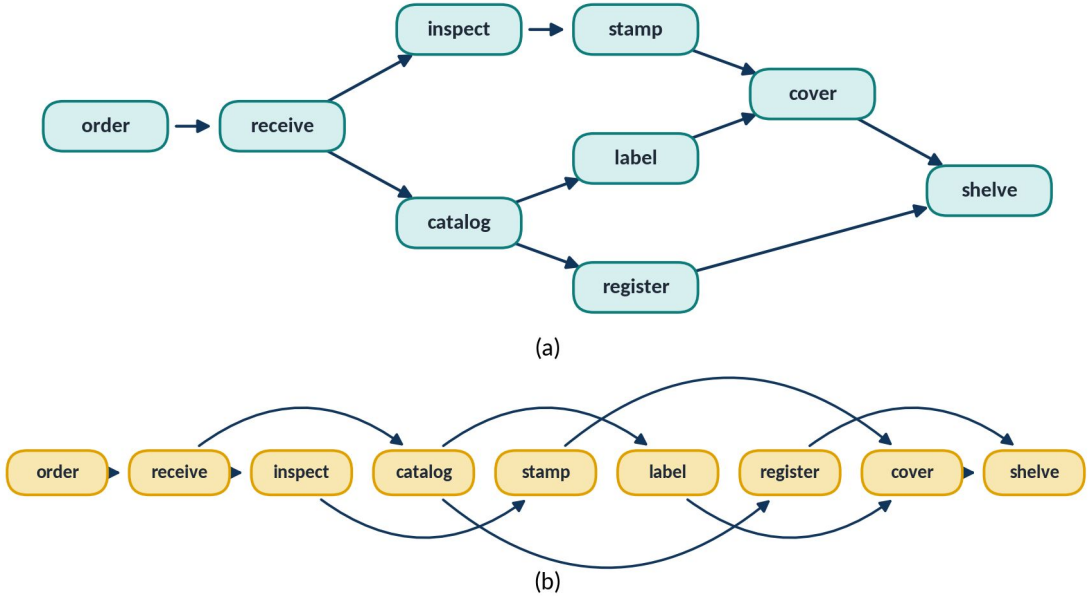


Figure 13.10 The new-book workflow: (a) the dependency DAG, with an arrow from each task to a task that depends on it; (b) the same DAG in the topological order found by Kahn's algorithm, where every arrow points to the right.

Figure 13.10: order, receive, inspect, catalog, stamp, label, cover, register, shelve; an arrow means "must come before". Laid out in a topological order, every arrow points right. Book: Section 13.9.1 and Figure 13.10.

Kahn's Algorithm

Pseudocode 13.5 KAHN: topological sort by repeatedly removing a vertex of in-degree 0.

```
KAHN(G):
 1.  for each vertex v in G.V
 2.      v.indeg ← 0
 3.  for each vertex u in G.V
 4.      for each v in G.adj[u]
 5.          v.indeg ← v.indeg + 1
 6.  Q ← a queue holding every vertex with indeg = 0
 7.  L ← an empty list
 8.  while Q is not empty
 9.      u ← DEQUEUE(Q)
10.      append u to L                 // u can be done now
11.      for each v in G.adj[u]        // cross u off: v loses a prerequisite
12.          v.indeg ← v.indeg - 1
13.          if v.indeg = 0
14.              ENQUEUE(Q, v)
15.  if L has fewer than |G.V| vertices
16.      report "the graph has a cycle"
17.  return L
```

Pseudocode 13.5: output a vertex of in-degree 0, again and again; $\Theta(V+E)$. Book: Sections 13.9.1–13.9.2.

Worked Example: Kahn on the Workflow

Table 13.3 Kahn's algorithm on the new-book workflow.

Step	Output u	In-degrees decreased	Reach 0 (enqueued)	Queue afterward
0			order	order
1	order	receive $1 \to 0$	receive	receive
2	receive	inspect $1 \to 0$, catalog $1 \to 0$	inspect, catalog	inspect catalog
3	inspect	stamp $1 \to 0$	stamp	catalog stamp
4	catalog	label $1 \to 0$, register $1 \to 0$	label, register	stamp label register
5	stamp	cover $2 \to 1$	none	label register
6	label	cover $1 \to 0$	cover	register cover
7	register	shelve $2 \to 1$	none	cover
8	cover	shelve $1 \to 0$	shelve	shelve
9	shelve	none	none	(empty)

Table 13.3: at step 5 cover still waits for label; it becomes free at step 6. Output: order, receive, inspect, catalog, stamp, label, register, cover, shelve. Book: Section 13.9.2.

Topological Sort by DFS

Pseudocode 13.6 `TOPOLOGICAL-SORT`: topological order by decreasing DFS finish time.

```
TOPOLOGICAL-SORT(G):
 1.  L ← an empty list
 2.  run DFS(G); each time a vertex u finishes (line 10 of DFS-VISIT),
 3.      insert u at the front of L
 4.  return L
```

For every arrow (u, v): v white becomes a descendant and finishes first; v black has already finished; v gray would be a cycle. So $v.f < u.f$.

Directed cycle $\Leftrightarrow$ an arrow into a **gray** vertex; an arrow into a black one is harmless. Three colors are needed.

Pseudocode 13.6 gives order, receive, catalog, register, label, inspect, stamp, cover, shelve: another valid order. Book: Sections 13.9.3–13.9.4.

A Mistaken Arrow Creates a Cycle

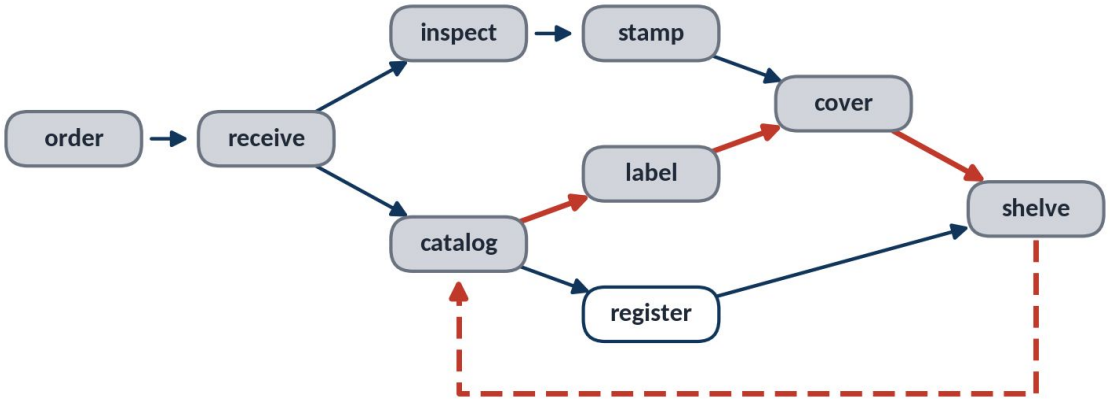


Figure 13.11 The workflow with a mistaken arrow from shelve to catalog (dashed). When DFS examines the arrow from label to cover, every task on the current path is gray and cover is one of them: a back edge, which proves the red cycle.

Figure 13.11: with shelve → catalog, Kahn outputs only 4 of 9 tasks and DFS finds the back edge label → cover. The detector proves a cycle; deciding which arrow is wrong is up to the librarian. Book: Section 13.9.4 and Figure 13.11.

Kahn in C++

Listing 13.3 `kahn` from `topo.cpp`: an order shorter than V means a cycle.

```
std::vector<int> kahn(const Graph &g, bool trace) {
    std::vector<int> indeg(V, 0), order;
    for (int u = 0; u < V; ++u)
        for (int v : g[u]) ++indeg[v];
    std::queue<int> q;
    for (int u = 0; u < V; ++u)
        if (indeg[u] == 0) q.push(u);
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        order.push_back(u);
        if (trace) std::printf("  output %-8s frees:", TASK[u]);
        for (int v : g[u]) {
            if (--indeg[v] == 0) {         // v's last prerequisite is done
                q.push(v);
                if (trace) std::printf(" %s", TASK[v]);
            }
        }
```

Listing 13.3; `make`, package managers and spreadsheets all sort topologically. Book: Section 13.9.5.

Weighted Shortest Paths: Relaxation

BFS counts edges; it cannot see that LIB, ADM, ENG (140 m) beats the direct LIB-ENG (150 m).

Keep a tentative distance $d[v]$, ∞ at first, and improve it by **relaxing** an edge, Equation (13.10):

$$\text{if } d[u]+w(u,v)<d[v]: d[v] \leftarrow d[u]+w(u,v), \pi[v] \leftarrow u \tag{13.10}$$

Dijkstra (1959): repeatedly take the not-yet-final vertex with the smallest d, declare it final, relax its edges. Weights must be nonnegative.

Relaxation never makes an estimate wrong: $d[v]$ is always the length of a real route, so it can only be too large. Book: Sections 13.10.1–13.10.3 and Equation (13.10).

Dijkstra with a Lazy Priority Queue

Pseudocode 13.7 DIJKSTRA: single-source shortest paths with nonnegative weights, using a min-priority queue with lazy deletion of stale entries.

```
DIJKSTRA(G, w, s):
 1.  for each vertex v in G.V
 2.      v.d ← ∞;  v.π ← NIL;  v.done ← FALSE
 3.  s.d ← 0
 4.  PQ ← an empty min-priority queue;  INSERT(PQ, (0, s))
 5.  while PQ is not empty
 6.      (du, u) ← EXTRACT-MIN(PQ)
 7.      if u.done                       // a stale entry: skip it
 8.          continue
 9.      u.done ← TRUE                   // u.d is now final
10.      for each v in G.adj[u]
11.          if not v.done and u.d + w(u, v) < v.d
12.              v.d ← u.d + w(u, v)     // relax (u, v)
13.              v.π ← u
14.              INSERT(PQ, (v.d, v))
```

Pseudocode 13.7: library queues cannot update an entry, so a better distance inserts a new one and the old stale entry is skipped later. With every weight 1 it behaves exactly like BFS. Book: Section 13.10.3.

Worked Example: Dijkstra from LIB

Table 13.4 Dijkstra's algorithm from LIB on the campus map. The priority queue is listed in increasing order of distance.

Step	Extracted	Status	Relaxations that improve d	Priority queue afterward
0			$d[\text{LIB}]=0$	(0, LIB)
1	(0, LIB)	final	CAF 120, ADM 80, ENG 150	(80, ADM) (120, CAF) (150, ENG)
2	(80, ADM)	final	ENG $150 \to 140$	(120, CAF) (140, ENG) (150, ENG)
3	(120, CAF)	final	DRM 320, MED 300	(140, ENG) (150, ENG) (300, MED) (320, DRM)
4	(140, ENG)	final	AUD 240	(150, ENG) (240, AUD) (300, MED) (320, DRM)
5	(150, ENG)	stale, skipped		(240, AUD) (300, MED) (320, DRM)
6	(240, AUD)	final	PRK 310 (MED: $400>300$, no)	(300, MED) (310, PRK) (320, DRM)
7	(300, MED)	final	none (DRM: $390>320$)	(310, PRK) (320, DRM)
8	(310, PRK)	final	none	(320, DRM)
9	(320, DRM)	final	none	(empty)

Table 13.4: at step 2, ADM improves ENG from 150 to 140; at step 5 the old (150, ENG) comes out stale. Book: Section 13.10.4.

Shortest Walking Distances

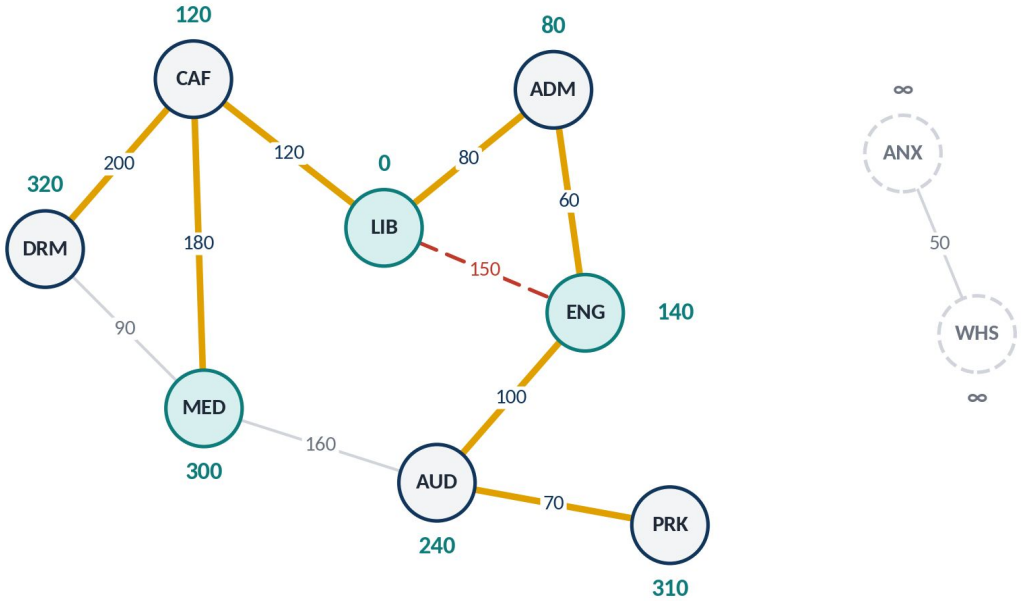


Figure 13.12 Shortest walking distances in meters from LIB. Amber edges form the shortest-path tree; the dashed red edge LIB-ENG is on the fewest-hop route to PRK but not on the shortest one.

Figure 13.12: PRK is 310 m away via ADM; the dashed red LIB-ENG is on the fewest-hop route but not on the shortest. Book: Section 13.10.4 and Figure 13.12.

Why It Works, and Its Cost

When u is extracted, a shorter route P would leave the set of finalized vertices at some y with $d[y] \le \text{length}(P) < d[u]$, so y would have come out first.

That step needs $w \ge 0$: with $w(s,a)=2$, $w(s,b)=3$, $w(b,a)=-2$, Dijkstra finalizes a at 2 and misses 1. Negative weights need Bellman-Ford.

$$T_{\text{Dijkstra}} = O((V+E)\log V) \tag{13.11}$$

Equation (13.11): everything non-linear comes from the priority queue; Chapter 14 opens that box. An array scan gives $\Theta(V^2)$, better for dense graphs. Book: Sections 13.10.5–13.10.6.

Dijkstra in C++

Listing 13.4 The loop of `dijkstra` in `dijkstra.cpp`; the queue `pq` is declared with `std::greater<Item>`, which turns the default max-heap into a min-heap.

```
    while (!pq.empty()) {
        auto [du, u] = pq.top();
        pq.pop();
        if (done[u]) {                      // an old, longer entry: ignore it
            std::printf("  pop (%3d, %s)  stale, skipped\n", du, CODE[u]);
            continue;
        }
        done[u] = true;                     // du is now final
        std::printf("  pop (%3d, %s)  final.", du, CODE[u]);
        for (const Nbr &e : adj[u]) {
            if (!done[e.to] && du + e.w < dist[e.to]) {   // relax u -> e.to
                dist[e.to] = du + e.w;
                parent[e.to] = u;
                pq.push({dist[e.to], e.to});
                std::printf(" %s=%d", CODE[e.to], dist[e.to]);
            }
        }
        std::printf("\n");
    }
```

Listing 13.4; Java's `PriorityQueue` and Python's `heapq` are min-heaps already. Book: Section 13.10.7.

Graphs at Real-World Scale

Table 13.5 Real graphs and the memory their two main representations would need. The OpenStreetMap segment count is estimated as nodes minus ways.

Graph	Vertices	Edges	Average degree	Matrix (1 bit per cell)	Adjacency list
Campus map (Section 13.4)	10	11	2.2	13 B	256 B
DIMACS USA roads, 2006 [14]	23,947,347	58,333,344 arcs	2.4 (out)	72 TB	658 MB
OpenStreetMap planet, Sep. 2026 [13]	10,882,374,688 nodes	≈ 9.66 billion segments	≈ 1.8	15 EB	242 GB
Meta family of apps, Jun. 2026 [16]	3.60 billion daily active people	not published	not published	≈ 1.6 EB	depends on edges

Table 13.5: OpenStreetMap held 10,882,374,688 nodes on September 23, 2026; Meta's apps had 3.60 billion daily active people in June 2026. A matrix of such graphs cannot exist; an adjacency list fits on a server. Book: Sections 13.11.1–13.11.2.

Matrix versus List, Plotted

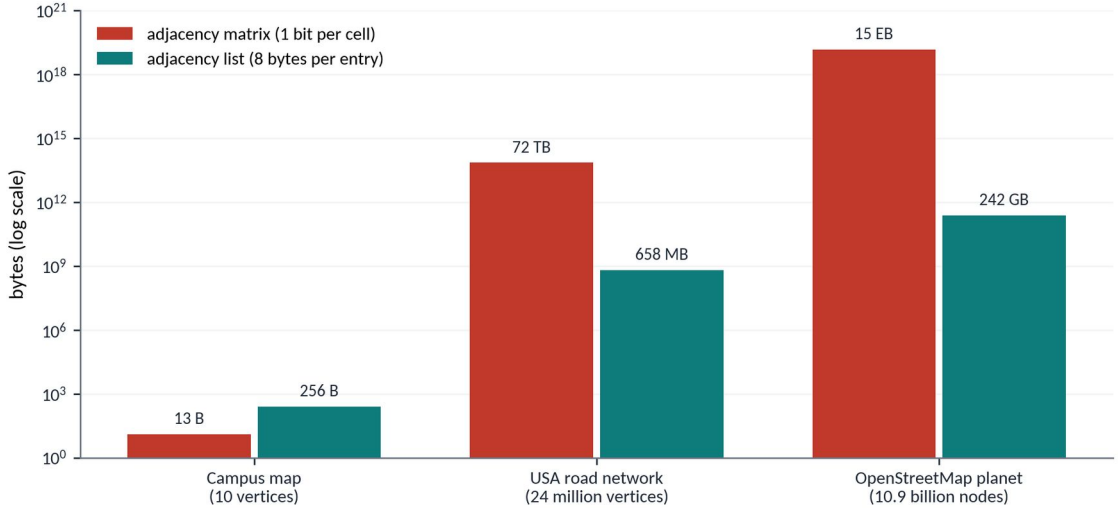


Figure 13.13 Memory for an adjacency matrix (one bit per cell) and an adjacency list (8 bytes per entry) of three road graphs: the campus map, the DIMACS USA network (2006 benchmark) and the OpenStreetMap planet (nodes and segments as of September 23, 2026). Logarithmic scale.

Figure 13.13: choose the representation by E, not V; aim for linear time; and on a road network a DFS goes millions of vertices deep, so production code uses an explicit stack. Book: Section 13.11.3 and Figure 13.13.

Choosing an Algorithm

Table 13.6 The graph algorithms of this chapter, with the structure each relies on and its cost on adjacency lists.

Question	Algorithm	Structure it relies on	Time	Extra memory
What is reachable from s?	BFS or DFS	queue or stack	$\Theta(V+E)$	$\Theta(V)$
Fewest edges from s to every vertex	BFS (Pseudocode 13.1)	FIFO queue	$\Theta(V+E)$	$\Theta(V)$
Discovery and finish times, DFS forest	DFS (Pseudocode 13.2, Pseudocode 13.3)	recursion or explicit stack	$\Theta(V+E)$	$\Theta(V)$
Does an undirected graph have a cycle?	DFS, back edge to a non-parent	stack	$\Theta(V+E)$	$\Theta(V)$
Connected components	one BFS or DFS per component (Pseudocode 13.4)	queue or stack	$\Theta(V+E)$	$\Theta(V)$
Order respecting all dependencies	Kahn (Pseudocode 13.5) or DFS finish times (Pseudocode 13.6)	queue, or stack	$\Theta(V+E)$	$\Theta(V)$
Does a directed graph have a cycle?	Kahn outputs fewer than V, or DFS finds a gray vertex	queue, or three colors	$\Theta(V+E)$	$\Theta(V)$
Shortest weighted routes, weights ≥ 0	Dijkstra (Pseudocode 13.7)	min-priority queue	$O((V+E)\log V)$	$O(V+E)$

Table 13.6: the same loop, "take a vertex out of a container and look at its neighbors", becomes BFS with a queue, DFS with a stack, and Dijkstra with a priority queue. Book: Section 13.12.

Profitina in Practice

A website is a **directed graph**: pages are vertices, hyperlinks are arrows; crawlers, including those of AI answer engines, follow them.

Reachable from the home page? One BFS; a page in the sitemap but not reachable is an *orphan*.

How many clicks deep? The hop distance. **Linked to most?** The in-degree. **Loops?** Of course; mark visited pages.

Profitina is an AI visibility (GEO) platform; we describe the shape of each question, not the product's implementation. Book: Section 13.13.

Worked Example: the Profitina Laundry Site

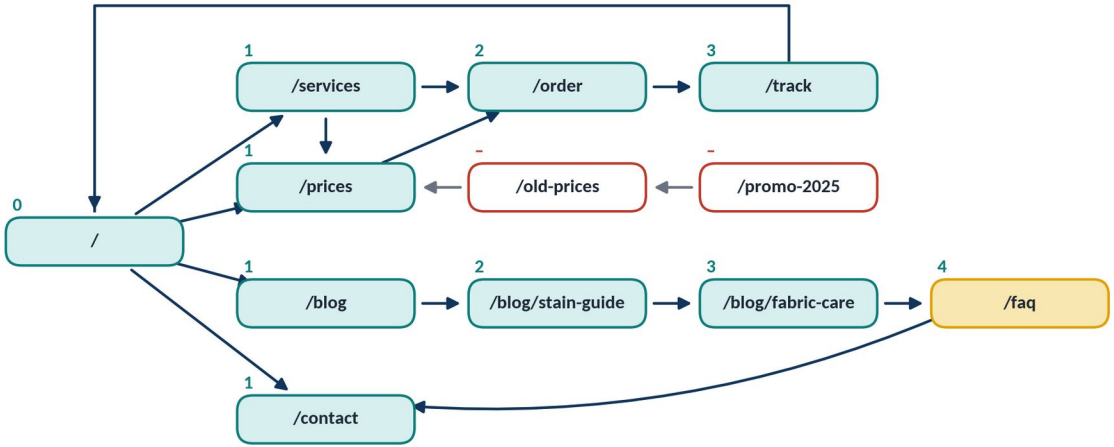


Figure 13.14 The Profitina Laundry website as a directed graph. The number above each page is its click depth from the home page; /faq (amber) is four clicks deep, and the two pages outlined in red are orphans that no link path reaches.

Figure 13.14: `/prices` has three inbound links; `/faq` is four clicks deep; `/promo-2025` and `/old-prices` are orphans. The program makes the structure visible; the fix is the owner's decision. Book: Section 13.13 and Figure 13.14.

Common Mistakes

Forgetting to mark visited vertices, or marking in BFS when a vertex *leaves* the queue.

Storing an undirected edge in one direction only; **a matrix for a large sparse graph**; **`list.pop(0)` as a Python queue**.

BFS for weighted routes; **Dijkstra with negative weights**.

Treating the parent as a cycle; **two colors for directed cycles**; **recursing too deep**; **relying on one particular visiting order**.

The handshake check catches a missing direction at once: the list lengths must add up to $2|E|$*. Book: Section 13.14.*

Summary: Five Things to Remember

1. $G=(V,E)$; degrees add up to $2|E|$; real graphs are sparse, so store them as adjacency lists.

2. BFS with a queue finds fewest-hop routes in $\Theta(V+E)$; mark on entry.

3. DFS with a stack gives discovery and finish times that nest like parentheses; back edges mean cycles.

4. Components: one search per unlabeled vertex. Topological order: Kahn's in-degrees, or decreasing finish times.

5. Dijkstra finalizes the closest vertex and relaxes its edges: $O((V+E)\log V)$ with a heap, nonnegative weights only.

Book: Section 13.15.

Next Time

Chapter 14 — Heaps and Priority Queues. The binary heap behind `std::priority_queue`, `PriorityQueue` and `heapq`: why insert and extract-min cost $O(\log n)$.

Decrease-key, heapsort, and Dijkstra with a real heap.

Before then: work Exercises 13.1, 13.4, 13.8 and 13.10, in C++ and then in Java or Python.

The solutions, with programs, are in the book. Book: Sections 13.16 and 13.17.

Lecture 14 — Heaps and Priority Queues

Course: Data Structure (EF234201)

Book: Irfan Subakti, *Data Structure with C++, Java and Python*, Chapter 14

Lecturer: Irfan Subakti — yifana@gmail.com

Today we open the box that Chapters 3, 5 and 13 used without looking inside: the binary heap, a tree that lives in an ordinary array. Book: Chapter 14, Sections 14.1–14.18.

What This Meeting Covers

1. **The priority queue** — serving the most urgent first, and why arrays and search trees are not enough.
2. **A complete tree in an array** — index formulas, height, where the leaves are; the heap property.
3. **Sift-up and sift-down** — and around them insert, extractMax, getMin, increaseKey, decreaseKey, isMaxHeap.
4. **Building in linear time, and heap sort in full.**
5. **Libraries and applications** — `std::priority_queue`, `PriorityQueue`, `heapq`; top-k, k-way merge, Dijkstra.

Running example: Pustaka Ceria hold requests, each with an urgency score; the circulation desk always serves the most urgent next. Book: Section 14.1.

Where We Left Off: Three Promises

Section 3.10 sorted with **heap sort** and promised the heap properly.

Section 5.15 met the **priority queue** (an emergency room serves the most urgent, not the earliest) and found that plain arrays make one operation slow.

Section 13.10 ran **Dijkstra** on a library priority queue and left open how it works.

Chapter 11 gave the vocabulary of trees, and even stored a binary tree in an array without pointers.

The binary heap needs no pointers: parent and children are a multiplication or a division by two away. About 20 steps for a million keys. Book: Section 14.1.

The Priority Queue

> **DEFINITION.** A *max-priority queue* is a collection of keys with `insert(x)`, `getMax()`, `extractMax()`, `increaseKey(i, k)` and `isEmpty()`. A *min-priority queue* has `getMin`, `extractMin` and `decreaseKey` instead.

Despite its name, it is not first-in, first-out: arrival order matters only to break ties, and only if we ask for it.

> **IN PLAIN LANGUAGE.** A priority queue never needs the whole collection sorted, only which item comes first. A heap keeps exactly that much order and no more.

A min-priority queue serves the smallest key: an earlier deadline, a shorter distance. Book: Section 14.2.1.

The Operations We Will Build

Table 14.1 The operations of the `MaxHeap` class of this chapter and their costs.

Operation	What it does	Cost on a heap of n keys	Section
`insert(x)`	add a key	$O(\lg n)$	14.7
`getMax()`	read the largest key	$O(1)$	14.7
`extractMax()`	remove the largest key	$O(\lg n)$	14.7
`getMin()`	read the smallest key	$\Theta(n)$	14.8
`increaseKey(i, k)`	raise the key at index i	$O(\lg n)$	14.9
`decreaseKey(i, k)`	lower the key at index i	$O(\lg n)$	14.9
`buildMaxHeap(a)`	make a heap of a whole array	$\Theta(n)$	14.10
`isMaxHeap(a)`	check the heap property	$O(n)$	14.11
`heapSort(a)`	sort an array in place	$O(n \lg n)$	14.13

***Table 14.1**: the costs are the point of the chapter, so we see them before we earn them. Book: Section 14.2.1.*

Why the Obvious Structures Fail

Table 14.2 Four ways to implement a max-priority queue of n keys. Only the binary heap is fast at every operation without extra memory.

Implementation	`insert`	`getMax`	`extractMax`	Extra memory
Unsorted array	$O(1)$	$\Theta(n)$	$\Theta(n)$	none
Sorted array	$\Theta(n)$	$O(1)$	$O(1)$	none
Balanced search tree	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$	two pointers per key
Binary heap	$O(\lg n)$	$O(1)$	$O(\lg n)$	none

The unsorted array does too little on arrival, the sorted array too much; the heap keeps **just enough** order, cheap to restore after every change.

***Table 14.2**. Williams published the binary heap in June 1964 with heapsort; six months later Floyd built it bottom-up in linear time. Sixty years later all three standard libraries still use it. Book: Sections 14.2.2–14.2.3.*

Complete Binary Trees

> **DEFINITION.** A binary tree of height h is *complete* when every level $0,1,\ldots,h-1$ is full, and the nodes of the last level h are packed as far left as possible, with no gaps.

A complete tree with n nodes is determined by n alone: number the nodes, and the number says where each one sits.

Cormen et al. call this shape *nearly complete*; the names differ, the shape is what matters.

The tree grows in exactly one place, the next free position of the last level. Book: Section 14.3.1.

Complete, and Not Complete

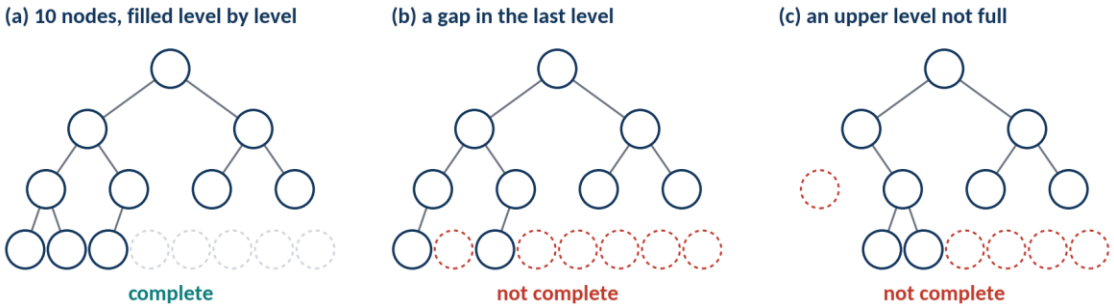


Figure 14.1 (a) A complete binary tree of ten nodes; dashed circles mark the free places of the last level, which fills from the left. (b) and (c) are not complete: (b) has a gap in its last level, and (c) has a missing node above the last level.

***Figure 14.1**: (b) has a gap in its last level; (c) is missing a node above the last level. Book: Section 14.3.1 and Figure 14.1.*

Numbering Level by Level

Number the nodes level by level, left to right, and store node i in array slot i. With the root at index 0, Equation (14.1):

$$\text{parent}(i) = \left\lfloor \frac{i-1}{2} \right\rfloor, \qquad \text{left}(i) = 2i + 1, \qquad \text{right}(i) = 2i + 2. \tag{14.1}$$

With the root at index 1 (Williams, Cormen et al.), Equation (14.2):

$$\text{parent}(k) = \left\lfloor \frac{k}{2} \right\rfloor, \qquad \text{left}(k) = 2k, \qquad \text{right}(k) = 2k + 1. \tag{14.2}$$

Substituting $k = i + 1$ turns one into the other. The proof counts the $2^d - 1$ nodes above level d; for a tree with gaps the formulas fail, so a heap must stay complete. Every program here uses 0-based indexes. Book: Sections 14.3.2–14.3.3.

Tree and Array, Side by Side

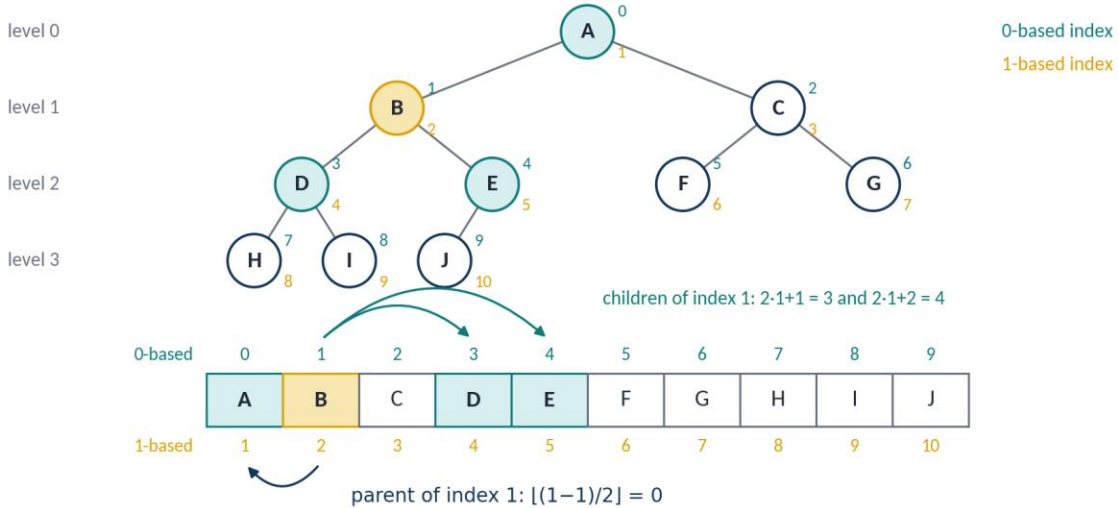


Figure 14.2 A complete tree stored level by level in an array. Each node carries its 0-based index (teal) and its 1-based index (amber); the arcs show that the children of index 1 are at indices 3 and 4, and its parent at index 0.

Figure 14.2*: B at index 1 has children D and E at 3 and 4, and parent A at 0; E at index 4 has only the left child J at 9. Book: Section 14.3.2 and Figure 14.2.*

The Index Arithmetic

Table 14.3 The index arithmetic of a complete binary tree of n nodes stored in an array.

Relative of the node	0-based (root at 0)	1-based (root at 1)
parent	$\lfloor (i-1)/2 \rfloor$	$\lfloor k/2 \rfloor$
left child	$2i+1$	$2k$
right child	$2i+2$	$2k+1$
is the root	$i = 0$	$k = 1$
has a left child	$2i+1 < n$	$2k \le n$
is a leaf	$i \ge \lfloor n/2 \rfloor$	$k > \lfloor n/2 \rfloor$

Table 14.3*: pick one numbering and use it everywhere; $2i$ and $2i+1$ in a 0-based array make the root its own left child. Book: Section 14.3.2.*

Height, and Where the Leaves Are

$$2^h \leq n \leq 2^{h+1} - 1. \tag{14.3}$$

$$h = \lfloor \lg n \rfloor. \tag{14.4}$$

$$\text{the internal nodes are } 0,1,\ldots,\lfloor n/2 \rfloor - 1, \quad \text{and the leaves are } \lfloor n/2 \rfloor,\ldots,n-1. \tag{14.5}$$

By Equation (14.3) and Equation (14.4), ten keys have height 3 and a million keys height 19. By Equation (14.5) the leaves are the second half of the array: at least half of all nodes. Book: Section 14.3.4.

The Heap Property

DEFINITION. An array $A[0..n-1]$, read as a complete binary tree, is a *max-heap* when every node other than the root is no larger than its parent, Equation (14.6), and a *min-heap* when it is no smaller, Equation (14.7).

$$A[\text{parent}(i)] \geq A[i] \qquad \text{for every } i = 1,2,\dots,n-1. \tag{14.6}$$

$$A[\text{parent}(i)] \leq A[i] \qquad \text{for every } i = 1,2,\dots,n-1. \tag{14.7}$$

The property compares only a node with its parent, yet by following parents up to the root, $A[0]$ is a largest key, and every root-to-leaf path is non-increasing. Book: Sections 14.4.1–14.4.2.

Max-Heap and Min-Heap

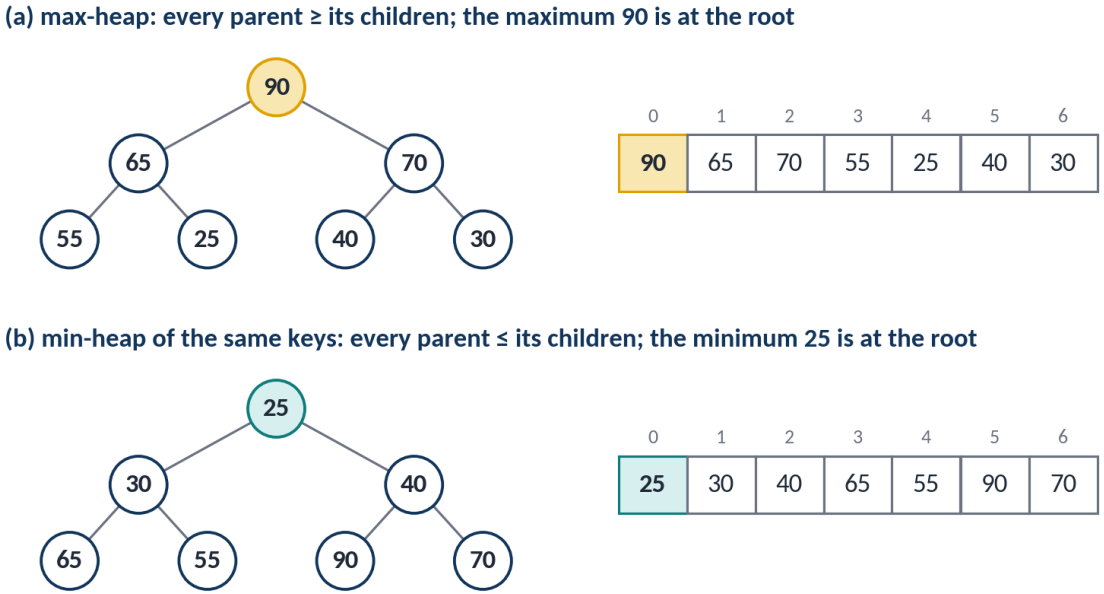


Figure 14.3 The keys 25, 30, 40, 55, 65, 70 and 90 arranged (a) as a max-heap, with the maximum at the root, and (b) as a min-heap, with the minimum at the root.

Figure 14.3*: the same seven keys; in the max-heap the path 90, 65, 55 decreases, and in the min-heap every comparison points the other way. Book: Section 14.4.1 and Figure 14.3.*

What the Heap Does Not Promise

Siblings are not ordered: 65 left of 70 is fine, and swapping the two subtrees gives another heap.

Levels are not ordered, and **the array is not sorted**: 90, 65, 70, 55, 25, 40, 30.

A heap is not a search tree: a key may be in either subtree, so finding an arbitrary key scans up to all n.

> **WARNING.** *Heap* also names the memory region from which `new` allocates (Section 1.3.5). The two meanings share only a name.

This weaker order is the price of speed: a partial order that is cheap to repair, enough to answer the one question a priority queue asks. Book: Section 14.4.3.

Sift-Up: a Key That Is Too Large

Pseudocode 14.1 `SIFT-UP`: let the key at index i rise until its parent is at least as large or it reaches the root.

```
SIFT-UP(A, i)                          // A[i] may be larger than its parent
 1  while i > 0 and A[parent(i)] < A[i]
 2      exchange A[i] with A[parent(i)]
 3      i = parent(i)
```

Invariant: the property holds for every $j \neq i$, and the parent of i is at least as large as the children of i.

Each iteration climbs one level: at most $\lfloor \lg n \rfloor$ exchanges and $\lfloor \lg n \rfloor + 1$ comparisons, $O(\lg n)$.

Pseudocode 14.1. *Also called heapify-up, bubble-up or swim. A refinement copies smaller parents down into a hole and writes the key once (Exercise 14.14). Book: Section 14.5.*

Worked Example: Sift-Up of 80

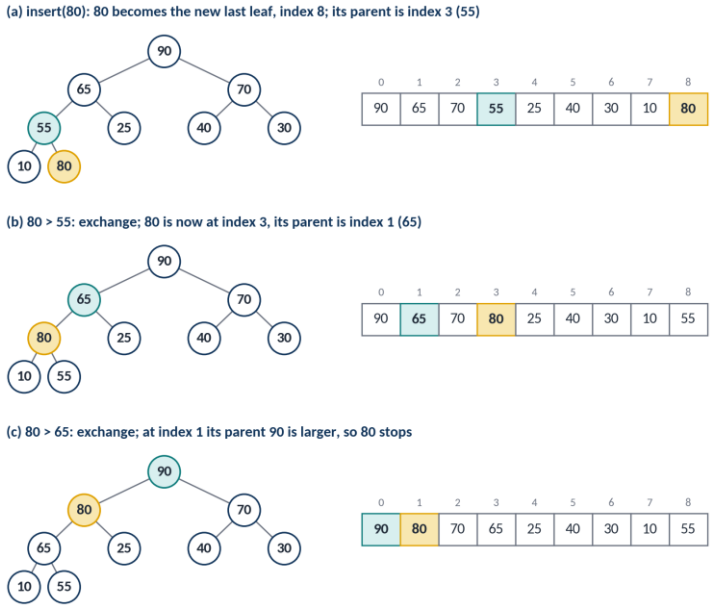


Figure 14.4 Sift-up of the key 80 from index 8. Each panel shows the tree and its array side by side; the moving key is amber and the parent it is compared with is teal.

Figure 14.4*: the parent of index 8 is index 3 (55), exchanged; then index 1 (65), exchanged; then the root 90 is larger, stop. Two exchanges, three comparisons. Book: Section 14.5.1 and Figure 14.4.*

Sift-Down: a Key That Is Too Small

Pseudocode 14.2 `SIFT-DOWN`: let the key at index i sink, exchanging it with its larger child, until no child is larger.

```
SIFT-DOWN(A, i, n)                  // heap is A[0..n-1]; A[i] may be too small
 1  loop
 2      largest = i
 3      l = 2i + 1;  r = 2i + 2
 4      if l < n and A[l] > A[largest]  then largest = l
 5      if r < n and A[r] > A[largest]  then largest = r
 6      if largest == i  then return        // A[i] ≥ both children
 7      exchange A[i] with A[largest]
 8      i = largest
```

Exchange with the **larger** child: it becomes at least as large as both the sinking key and its sibling.

$$\text{comparisons of SIFT-DOWN} \le 2\lfloor \lg n \rfloor, \qquad \text{exchanges} \le \lfloor \lg n \rfloor. \tag{14.8}$$

Pseudocode 14.2 *takes the size n as a parameter, because heap sort sifts inside the front part of an array. Equation (14.8): twice the comparisons of sift-up, two candidates per level. From a node of height h: at most h exchanges. Book: Section 14.6.*

Worked Example: Sift-Down of 55

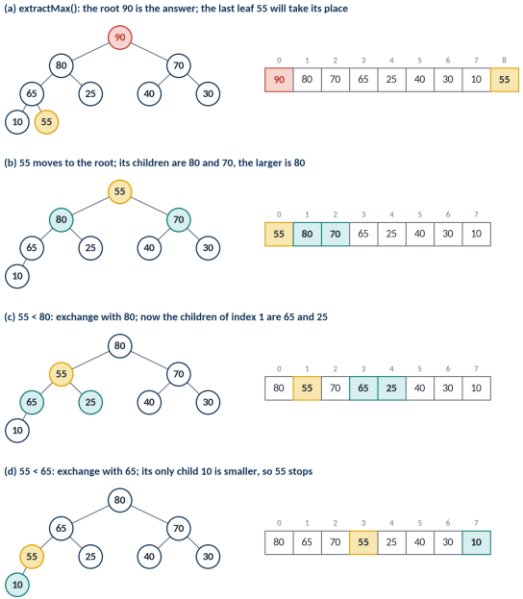


Figure 14.5 Sift-down of the key 55 from the root during `extractMax`: at each level it is compared with both children (teal) and exchanged with the larger one, until no child is larger.

Figure 14.5*: children 80 and 70, exchange with 80; children 65 and 25, exchange with 65; only child 10 is smaller, stop. Book: Section 14.6.1 and Figure 14.5.*

The Two Repairs in C++

Listing 14.1 `siftUp` and `siftDown` from `max_heap.cpp`.

```
    // Lets heap[i] rise while it is larger than its parent.
    void siftUp(int i) {
        while (i > 0 && heap[parent(i)] < heap[i]) {
            swap(heap[i], heap[parent(i)]);
            i = parent(i);
        }
    }

    // Lets a[i] sink inside the heap a[0..n-1] until no child is larger.
    static void siftDown(vector<int>& a, int i, int n) {
        while (true) {
            int largest = i, l = left(i), r = right(i);
            if (l < n && a[l] > a[largest]) largest = l;
            if (r < n && a[r] > a[largest]) largest = r;
            if (largest == i) return;
            swap(a[i], a[largest]);
            i = largest;
        }
    }
```

***Listing 14.1**: `l < n` and `r < n` guard the children that may not exist. Every other operation is a few lines around one of these two loops. Book: Section 14.12.2.*

`insert`: Append, Then Rise

Pseudocode 14.3 `INSERT`: append the key as the last leaf, then let it rise.

```
INSERT(A, n, x)                    // A[0..n-1] is a max-heap
 1  A[n] = x                       // the new last leaf keeps the tree complete
 2  n = n + 1
 3  SIFT-UP(A, n - 1)
```

```
insert(40)  -> 40
insert(25)  -> 40 25
insert(70)  -> 70 25 40
insert(10)  -> 70 25 40 10
insert(55)  -> 70 55 40 10 25
insert(90)  -> 90 55 70 10 25 40
```

Pseudocode 14.3*: the new last leaf keeps the tree complete. Work* `insert(90)` *by hand: 90 at index 5, exchanged with 40, then with 70; it reaches the root. O*(*lgn*)*. Book: Section 14.7.2.*

`extractMax`: the Last Leaf Moves Up

Pseudocode 14.4 `EXTRACT-MAX`: save the root, move the last leaf into the root, shrink the heap, and let the new root sink.

```
EXTRACT-MAX(A, n)                    // A[0..n-1] is a max-heap, n ≥ 1
 1  if n == 0  then error "the heap is empty"
 2  top = A[0]
 3  A[0] = A[n - 1]                  // the last leaf fills the hole at the root
 4  n = n - 1
 5  if n > 0  then SIFT-DOWN(A, 0, n)
 6  return top
```

Only removing the *last* leaf keeps the tree complete; so move it into the root and let it sink.

On the nine hold requests: returns 90, then 80; the heap becomes 70, 65, 40, 55, 25, 10, 30.

Pseudocode 14.4*: shape first, by moving the last leaf; order second, by sifting. That is the method behind every heap operation. At most* $2\lfloor lgn \rfloor$ *comparisons. Book: Section 14.7.3.*

`insert` and `extractMax` in C++

Listing 14.2 `insert`, `getMax` and `extractMax` from `max_heap.cpp`.

```
    // Appends key as the last leaf, then lets it rise: O(lg n).
    void insert(int key) {
        heap.push_back(key);
        siftUp(size() - 1);
    }

    // Returns the largest key without removing it: O(1).
    int getMax() const {
        requireNonEmpty("getMax");
        return heap[0];
    }

    // Removes and returns the largest key: O(lg n).
    int extractMax() {
        requireNonEmpty("extractMax");
        int top = heap[0];
        heap[0] = heap.back();              // the last leaf fills the hole
        heap.pop_back();
        if (!heap.empty()) siftDown(heap, 0, size());
        return top;
    }
```

***Listing 14.2**: an empty heap throws `std::underflow_error` rather than return a made-up value. Book: Sections 14.7 and 14.12.*

`getMin` in a Max-Heap

Pseudocode 14.5 `GET-MIN` for a max-heap: the minimum is on a leaf, so scan the leaves, the second half of the array.

```
GET-MIN(A, n)                          // A[0..n−1] is a max-heap, n ≥ 1
 1  smallest = A[⌊n/2⌋]                // the first leaf
 2  for i = ⌊n/2⌋ + 1 to n − 1
 3      if A[i] < smallest  then smallest = A[i]
 4  return smallest
```

Some leaf holds a smallest key: follow children down from it, each no larger. So scan the leaves, the second half.

No method does better: lower any leaf below every other key and the array is still a max-heap, so every leaf must be looked at: $\Omega(n)$.

***Pseudocode 14.5**. When both extremes are needed quickly: two heaps side by side, a min-max heap, or a balanced search tree. Book: Section 14.8.*

The Leaves of Nine Keys

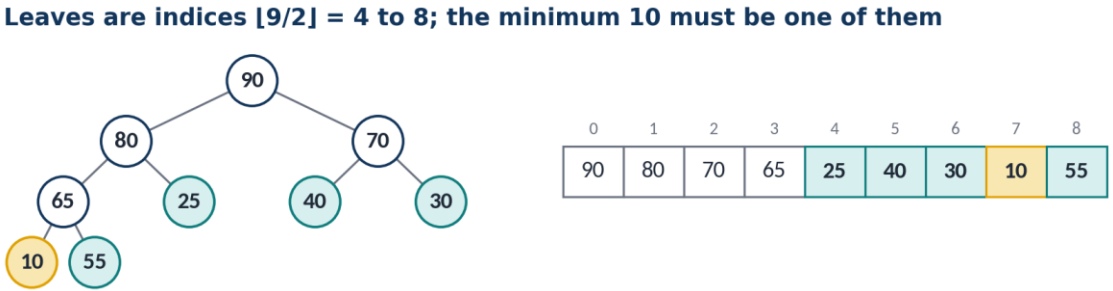


Figure 14.6 The leaves of a max-heap of nine keys are the indices 4 to 8, the second half of the array (teal); the minimum, 10 (amber), is one of them.

***Figure 14.6**: indices $\lfloor 9/2 \rfloor = 4$ to 8 hold 25, 40, 30, 10, 55; four comparisons find 10. Book: Section 14.8 and Figure 14.6.*

`increaseKey` and `decreaseKey`

Pseudocode 14.6 `INCREASE-KEY` and `DECREASE-KEY` in a max-heap: change the key, then let it rise or sink.

```
INCREASE-KEY(A, n, i, k)              // raise A[i] to k
 1  if i < 0 or i ≥ n  then error "index is outside the heap"
 2  if k < A[i]  then error "new value is below the key"
 3  A[i] = k
 4  SIFT-UP(A, i)

DECREASE-KEY(A, n, i, k)              // lower A[i] to k
 1  if i < 0 or i ≥ n  then error "index is outside the heap"
 2  if k > A[i]  then error "new value is above the key"
 3  A[i] = k
 4  SIFT-DOWN(A, i, n)
```

Pseudocode 14.6*: raising a key can only make it too large, lowering too small; one assignment and one repair, $O(\lg n)$. Reject a change in the wrong direction. The caller must know the index, which moves: an indexed priority queue keeps a second table. Book: Section 14.9.*

Worked Example: 75 Rises, 20 Sinks

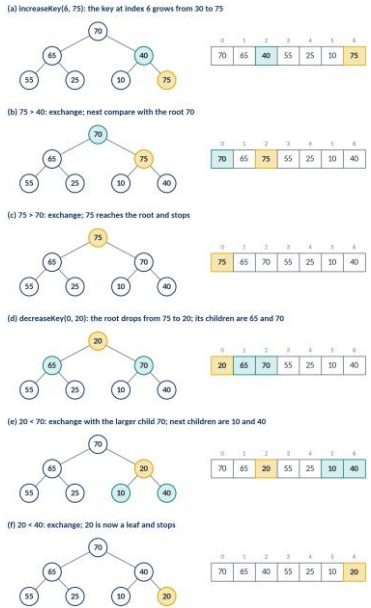


Figure 14.7 (a) to (c) `increaseKey(6, 75)`: the raised key rises by sift-up. (d) to (f) `decreaseKey(0, 20)`: the lowered key sinks by sift-down, always toward the larger child.

Figure 14.7: *`increaseKey(6, 75)` gives 75, 65, 70, 55, 25, 10, 40; `decreaseKey(0, 20)` gives 70, 65, 40, 55, 25, 10, 20. Book: Section 14.9 and Figure 14.7.*

Building Bottom-Up

Pseudocode 14.7 `BUILD-MAX-HEAP`: sift down every internal node, from the last one back to the root.

```
BUILD-MAX-HEAP(A, n)
 1  for i = ⌊n/2⌋ − 1 downto 0        // every internal node, bottom-up
 2      SIFT-DOWN(A, i, n)
```

The leaves are already one-node heaps; sift down every internal node from $\lfloor n/2 \rfloor - 1$ back to the root.

Invariant: at the iteration for i, every node $i+1, \ldots, n-1$ is the root of a max-heap, exactly sift-down's precondition.

Pseudocode 14.7 *is Floyd's method (1964). Start at* $\lfloor n/2 \rfloor - 1$*, not at* $n - 1$ *and not at 0 going upward. Book: Section 14.10.1.*

Worked Example: `buildMaxHeap`

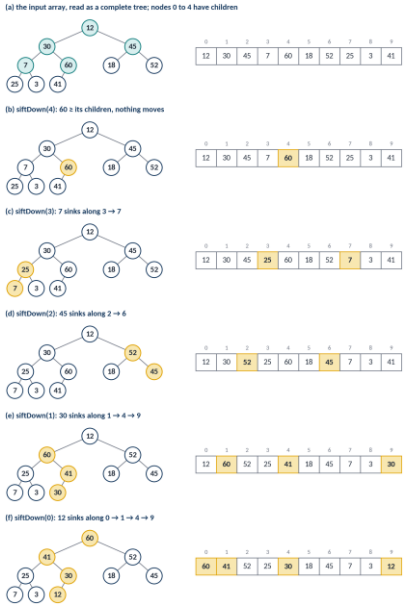


Figure 14.8 `buildMaxHeap` on 12, 30, 45, 7, 60, 18, 52, 25, 3, 41: sift-down is applied to the internal nodes 4, 3, 2, 1 and 0 in that order; amber marks the path along which each key sinks.

***Figure 14.8**: nodes 4, 3, 2, 1, 0 in that order; the root 12 sinks past 60, 41 and 30. Result 60, 41, 52, 25, 30, 18, 45, 7, 3, 12, after seven exchanges. Book: Section 14.10.1 and Figure 14.8.*

Why the Build Takes Linear Time

$$S = \sum_{d=0}^{H} 2^d\,(H-d) = \sum_{h=0}^{H} h\;2^{H-h} = 2^H \sum_{h=0}^{H} \frac{h}{2^h}, \tag{14.9}$$

$$T - \frac{T}{2} = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \cdots = 1, \qquad \text{so} \qquad \sum_{h=0}^{\infty} \frac{h}{2^h} = T = 2. \tag{14.10}$$

$$E_{\text{build}}(n) < n, \qquad C_{\text{build}}(n) < 2n, \qquad T_{\text{build}}(n) = \Theta(n), \tag{14.11}$$

Half the nodes are leaves, a quarter can sink one level, an eighth two. Equation (14.9) with Equation (14.10) gives $S < n + 1$; exactly $n - \lg(n+1)$ for a perfect heap, 11 for 15 nodes. Hence Equation (14.11). Book: Section 14.10.2.

Heights in a 15-Node Heap

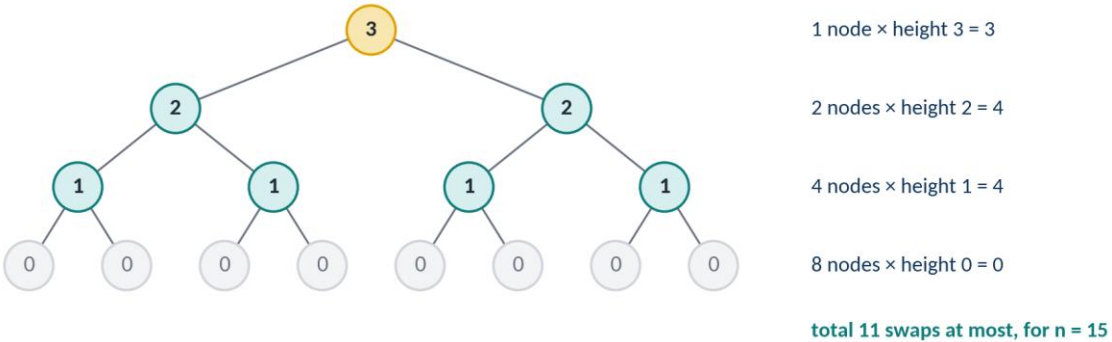


Figure 14.9 Each node of a 15-node heap labeled with its height, the most levels a sift-down from it can travel. Summing level by level gives at most 11 exchanges for the whole build, fewer than the 15 nodes.

Figure 14.9: $8 \cdot 0 + 4 \cdot 1 + 2 \cdot 2 + 1 \cdot 3 = 11$ *exchanges at most. The work is done where there is little to do. Book: Section 14.10.2 and Figure 14.9.*

Building by Insertion Is Really Slower

$$\sum_{m=2}^{n} \lfloor \lg m \rfloor = (n+1)\lfloor \lg n \rfloor - 2^{\lfloor \lg n \rfloor + 1} + 2 = \Theta(n \lg n). \tag{14.12}$$

input	n	n inserts	per n	build	per n
ascending	1000	7987	7.99	1982	1.98
random	1000	2236	2.24	1873	1.87
descending	1000	999	1.00	999	1.00
ascending	10000	113631	11.36	19982	2.00
random	10000	22788	2.28	18857	1.89
descending	10000	9999	1.00	9999	1.00
ascending	100000	1468946	14.69	199978	2.00
random	100000	228358	2.28	188116	1.88
descending	100000	99999	1.00	99999	1.00

Equation (14.12): ascending keys each climb to the root, 14.69 comparisons per key at $n = 100{,}000$; the bottom-up build stays at or below 2.00. Random insertions average about 2.28 per key. Book: Section 14.10.3.

`isMaxHeap`: Checking the Property

Pseudocode 14.8 `IS-MAX-HEAP`: check every child against its parent; an empty or one-element array is a heap.

```
IS-MAX-HEAP(A, n)
 1  for i = 1 to n - 1
 2      if A[parent(i)] < A[i]  then return false
 3  return true
```

In 10, 7, 8, 5, 2, 3, 1 every node is no larger than its parent; in 10, 7, 6, 5, 1, 8, 2 the key 8 sits under 6. A `true` answer must have checked all $n - 1$ pairs.

Pseudocode 14.8 *loops over children, so the empty and one-element arrays need no special case; a loop bound like* `(v.size() - 2) / 2` *wraps around in C++. Book: Section 14.11.*

The `MaxHeap` Class

Table 14.4 The public interface of the `MaxHeap` class in C++, Java and Python.

C++ and Java	Python	Cost	On an empty heap or a bad argument
`insert(key)`	`insert(key)`	$O(\lg n)$	never fails
`getMax()`	`get_max()`	$O(1)$	empty: error
`extractMax()`	`extract_max()`	$O(\lg n)$	empty: error
`getMin()`	`get_min()`	$\Theta(n)$	empty: error
`increaseKey(i, v)`	`increase_key(i, v)`	$O(\lg n)$	bad index or v below the key: error
`decreaseKey(i, v)`	`decrease_key(i, v)`	$O(\lg n)$	bad index or v above the key: error
`buildMaxHeap(a)`	`build_max_heap(a)`	$\Theta(n)$	replaces the contents
`isMaxHeap(a)` (static)	`is_max_heap(a)` (static)	$O(n)$	true for 0 or 1 keys
`heapSort(a)` (static)	`heap_sort(a)` (static)	$O(n \lg n)$	sorts `a` in place
`size()`, `empty()`, `clear()`	`size()`, `empty()`, `clear()`	$O(1)$	—
`printHeap()`, `data()`	`print_heap()`, `data()`	$O(n)$	—

***Table 14.4**; errors are exceptions. Book: Section 14.12.*

Heap Sort in Full

Pseudocode 14.9 `HEAP-SORT`: build a max-heap, then repeatedly move the root to the end of the shrinking heap and repair the root.

```
HEAP-SORT(A, n)
 1  BUILD-MAX-HEAP(A, n)
 2  for end = n − 1 downto 1
 3      exchange A[0] with A[end]      // the largest remaining key to its place
 4      SIFT-DOWN(A, 0, end)           // repair the heap A[0..end−1]
```

Invariant: $A[0..end]$ is a max-heap of the $end + 1$ smallest keys; $A[end + 1..n - 1]$ holds the largest keys in ascending order.

Pseudocode 14.9*: phase 2 is* `extractMax` *that parks the maximum in the slot the shrinking heap just gave up. Book: Sections 14.13.1–14.13.2.*

Worked Example: Heap Sort of Ten Keys

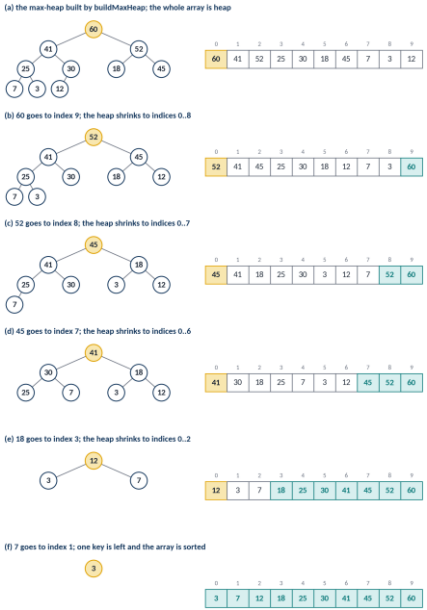


Figure 14.10 Heap sort on the heap of **Figure 14.8**: after each exchange and sift-down the heap (the tree, root in amber) shrinks by one and the sorted tail (teal) grows by one; panels (e) and (f) skip intermediate steps.

Figure 14.10*: 60 goes to index 9, then 52 to index 8, and so on; the program counts 38 comparisons. Book: Section 14.13.1 and Figure 14.10.*

What Heap Sort Costs

$$C_{\text{heap sort}}(n) \leq 2n + 2(n-1)\lfloor \lg n \rfloor = O(n \lg n) \tag{14.13}$$

Table 14.5 Heap sort compared with the other $O(n \lg n)$ comparison sorts of Chapter 3.

Sort	Worst case	Average	Extra memory	Stable
Merge sort (Section 3.8)	$O(n \lg n)$	$O(n \lg n)$	$\Theta(n)$	yes
Quicksort (Section 3.9)	$\Theta(n^2)$	$O(n \lg n)$	$O(\lg n)$ stack	no
Heap sort (this section)	$O(n \lg n)$	$O(n \lg n)$	$O(1)$	no

*Equation (14.13) in every case, $O(1)$ extra memory: the only sort of Chapter 3 with both a worst-case guarantee and sorting in place (**Table 14.5**). Poor cache use makes it slower than a good quicksort; introsort switches to it only when recursion grows suspiciously deep. Book: Section 14.13.3.*

Why Heap Sort Is Not Stable

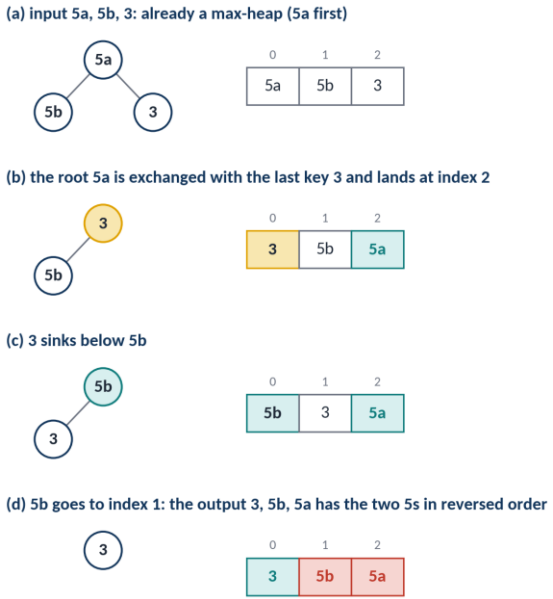


Figure 14.11 Heap sort of 5a, 5b, 3 by key: the first exchange moves 5a from the front to the back, past 5b, and the output ends with 5b before 5a.

***Figure 14.11**: the first exchange throws 5a from the front to the back, past 5b. Sorting books by year puts Amba before Pulang; `std::stable_sort` does not. Book: Section 14.13.4 and Figure 14.11.*

Priority Queues in the Libraries

Table 14.6 The priority queues of the three standard libraries. All three are binary heaps in arrays.

	C++ `std::priority_queue`	Java `PriorityQueue`	Python `heapq`
Default order	max first	min first	min first (max functions since 3.14)
Read the top	`top()`	`peek()`	`h[0]`
Insert	`push(x)`	`add(x)`, `offer(x)`	`heappush(h, x)`
Remove the top	`pop()` (returns `void`)	`poll()`	`heappop(h)`
Build from n items	constructor, `make_heap`	constructor from a collection	`heapify(h)`
Custom order	`Compare` type (“served later”)	`Comparator`	tuple keys, or negated keys
Empty top	undefined behavior	`peek` returns `null`	`IndexError`
`decreaseKey`	no	no (only `remove` in $O(n)$)	no

***Table 14.6**: `std::priority_queue` is a max-heap whose comparison means “is served later than”, and `pop()` returns nothing; Java’s `PriorityQueue` is a min-heap; none offers `decreaseKey`. Book: Sections 14.14.1–14.14.2.*

Python's `heapq`, and Ties

Listing 14.3 From `library_pq.py`: tuples `(-urgency, arrival, member, title)` in a min-heap.

```
requests = [(70, 1, "Ayu", "Laskar Pelangi"), (90, 2, "Budi", "Bumi Manusia"),
            (40, 3, "Citra", "Saman"), (90, 4, "Dewi", "Pulang"),
            (55, 5, "Eko", "Amba"), (70, 6, "Fajar", "Cantik Itu Luka")]

holds = []                                  # a min-heap of tuples, compared in order
for urgency, arrival, member, title in requests:
    heapq.heappush(holds, (-urgency, arrival, member, title))

print(f"Serving {len(holds)} hold requests (urgency, then arrival):")
while holds:
    neg, arrival, member, title = heapq.heappop(holds)   # O(lg n)
    print(f"  urgency {-neg:2d}  #{arrival}  {member:<6} {title}")
```

***Listing 14.3**: tuples compare left to right, so the arrival number breaks ties and keeps records from ever being compared by name. Python 3.14 added `heapify_max`, `heappush_max`, `heappop_max`, `heappushpop_max` and `heapreplace_max`. Book: Section 14.14.3.*

Top-k of a Stream

Pseudocode 14.10 `TOP-K`: keep the k best items in a min-heap whose root is the weakest of them.

```
TOP-K(stream, k)                    // the k largest items, using a min-heap
 1  H = empty min-heap
 2  for each item x in stream
 3      if size(H) < k  then INSERT(H, x)
 4      else if x > MIN(H)
 5          replace the root of H by x; SIFT-DOWN at the root
 6  return the items of H             // sort them for display if needed
```

$$T_{\text{top-}k}(n) = O(n \lg k) \text{ time}, \qquad O(k) \text{ memory}. \tag{14.14}$$

Pseudocode 14.10*: a* ***min****-heap of size k whose root is the weakest of the best; an item that does not beat it is discarded at once. Equation (14.14): it works on a stream too large to keep. Book: Section 14.15.1.*

Top 3, and a Three-Way Merge



Figure 14.12 (a) Top 3 of a stream: when 33 arrives, the min-heap holds 27, 31 and 40; since 33 > 27, it replaces the root. Teal marks the top 3 at the end of the stream. (b) Merging three sorted lists: the min-heap holds the front of each list (amber), and the root is always the next output.

***Figure 14.12**: the weakest kept rises 8, 14, 19, 27, 31; the top 3 are 40, 33, 31. The merge always outputs the root. Book: Sections 14.15.1–14.15.2 and Figure 14.12.*

Merging k Sorted Lists

Pseudocode 14.11 `MERGE-K`: merge k sorted lists with a min-heap that holds one front element from each list.

```
MERGE-K(L_1, ..., L_k)                // each L_j is sorted
 1  H = empty min-heap of (value, list j, position)
 2  for j = 1 to k
 3      if L_j is not empty  then INSERT(H, (L_j[0], j, 0))
 4  while H is not empty
 5      (v, j, p) = EXTRACT-MIN(H)
 6      output v
 7      if p + 1 < length(L_j)  then INSERT(H, (L_j[p + 1], j, p + 1))
```

$$T_{\text{merge}}(N, k) = O(N \lg k). \tag{14.15}$$

***Pseudocode 14.11**, Equation (14.15): the heap never holds more than k entries. The same idea sorts data too large for memory: sorted runs merged with one heap (external sorting). Book: Section 14.15.2.*

Top-k in C++

Listing 14.4 Part 1 of `topk_merge.cpp`: a `priority_queue` with `greater` keeps the three best borrow counts.

```
    const size_t k = 3;
    priority_queue<pair<int, string>, vector<pair<int, string>>,
                   greater<pair<int, string>>> top;       // smallest on top
    for (const auto& b : borrows) {
        if (top.size() < k) top.push(b);
        else if (b.first > top.top().first) {             // beats the weakest
            top.pop();
            top.push(b);
        }
        printf("read %2d %-21s -> weakest kept: %d\n", b.first, b.second.c_str(),
               top.top().first);
    }
```

***Listing 14.4**: the result is checked against a full sort. Python's `heapq.nlargest` and `heapq.merge` implement the same two algorithms. Book: Section 14.15.*

The Hook to Chapter 13: Dijkstra

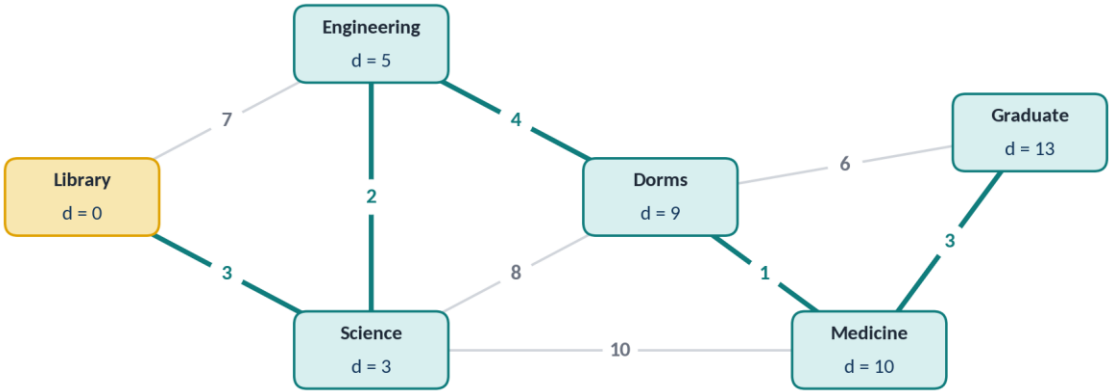


Figure 14.13 The delivery roads with travel times in minutes, and the shortest delivery time d from the Library to each building. The teal edges form the shortest-path tree that the priority queue finds.

***Figure 14.13**: lowering $d[v]$ is a `decreaseKey`; without it, the lazy version pushes a new entry and skips the stale one later. Four of ten entries were stale. Book: Section 14.15.3 and Figure 14.13.*

Dijkstra's Cost, Explained

$$T_{\text{Dijkstra}} = O((|V| + |E|)\lg|V|). \tag{14.16}$$

At most one push per edge direction: the heap holds $O(|E|)$ entries, each push or pop $O(\lg|V|)$ since $|E| < |V|^2$. An indexed heap with true `decreaseKey` keeps at most $|V|$ entries; the Fibonacci heap gives $O(|E| + |V|\lg|V|)$.

Equation (14.16): on ordinary graphs the simple binary heap with lazy deletion is usually the fastest in practice. Book: Section 14.15.3.

Profitina in Practice

What is due next: automatic re-scans at different cadences form a min-priority queue keyed by due time.

The top of a leaderboard: top-k with a min-heap of size k, without sorting everyone.

A dashboard merged from many feeds: a k-way merge that can stop after the newest 20.

A crawl frontier: a priority queue by importance; a FIFO queue is plain BFS.

The server is Python with FastAPI, so `heapq` is the heap at hand; we describe the shapes, not the product's implementation. In every case a heap replaces a repeated full scan or sort by a logarithmic step. Book: Section 14.16.

Common Mistakes

Mixing 0-based and 1-based formulas; **sifting toward the wrong child**; **reading a child that does not exist**.

Removing the root without moving the last leaf; **assuming a heap is sorted**; **searching a heap like a search tree**.

Building by n insertions when all keys are known; **changing a key without repairing**.

A backward `std::priority_queue` comparison; **leaving ties to chance**; **unsigned loop bounds**.

And confusing the data structure with the memory region of the same name. Book: Section 14.17.

Summary: Five Things to Remember

1. A complete tree lives in an array: children $2i + 1$, $2i + 2$, parent $\lfloor (i - 1)/2 \rfloor$, height $\lfloor \lg n \rfloor$.

2. Max-heap: every node no larger than its parent, so the root is the maximum; nothing else is ordered.

3. Sift-up and sift-down repair one bad key in $O(\lg n)$; insert, extractMax and key changes are built on them.

4. Build bottom-up in $\Theta(n)$; heap sort in $O(n \lg n)$ with $O(1)$ memory, but not stable.

5. Libraries: C++ max first, Java and Python min first; top-k, k-way merge and Dijkstra all run on a heap.

Book: Section 14.18.

Next Time

Chapter 15 — Hashing and Hash Tables. Find a key in expected $O(1)$ time by computing where it lives.

Hash functions, collisions, chaining and open addressing, load factor and rehashing.

Before then: work Exercises 14.3, 14.5, 14.8 and 14.11, in C++ and then in Java or Python.

The solutions, with programs, are in the book. Book: Sections 14.19 and 14.20.

Lecture 15 — Hashing and Hash Tables

Course: Data Structure (EF234201)

Book: Irfan Subakti, *Data Structure with C++, Java and Python*, Chapter 15

Lecturer: Irfan Subakti — yifana@gmail.com

Today we stop comparing keys and start computing where they live: search, insert and delete in constant time on average. Book: Chapter 15, Sections 15.1–15.14.

What This Meeting Covers

1. The idea — from $O(n)$ to $O(\lg n)$ to $O(1)$; direct addressing; the vocabulary.

2. Hash functions — division, multiplication, polynomial string hashes, the rolling update, avalanche.

3. Collisions — the birthday bound; separate chaining; open addressing with tombstones.

4. Load factor and rehashing, measured; worst cases and hash flooding.

5. The libraries — `std::unordered_map` versus `std::map`, Java's `HashMap`, Python's `dict`; applications.

We build two tables by hand, measure them, break one on purpose, and then read what the standard libraries actually do. Book: Section 15.1.

Where We Left Off: an Escape Clause

Section 2.12.2: no algorithm that only **compares** keys beats $\lceil \lg(n+1) \rceil$ probes in the worst case.

Balanced search trees (Chapter 11) reach $O(\lg n)$, but they are comparison machines too.

An algorithm that does **arithmetic** on the key is not bound by the proof. We already met it: 256 counters (Section 4.4.3), a set of seen values (8.7.3), a map from title to node (9.9).

The price: no order (no sorted listing, no range query, no successor), and the constant time is an **average**, not a guarantee.

Section 15.8 shows how bad the worst case gets and how real systems defend themselves. Book: Section 15.1.

The Librarian Who Never Searches

The volunteer looks through the pile one by one: $O(n)$. The student keeps the books sorted and halves: $O(\lg n)$.

The head librarian has 26 trays, A to Z, and puts each book in the tray of its first letter; asked about *Filosofi Kopi*, she reaches into tray F.

The rule is a **hash function**, the trays are **buckets**, *Fisika Dasar* and *Filosofi Kopi* in the same tray are a **collision**.

A thousand books titled "Sejarah …" would bury tray S. Everything in this chapter refines the trays: a better rule, the right number of trays, a plan for shared trays, and more trays when the pile grows. Book: Section 15.2.1.

Direct Addressing and Hashing

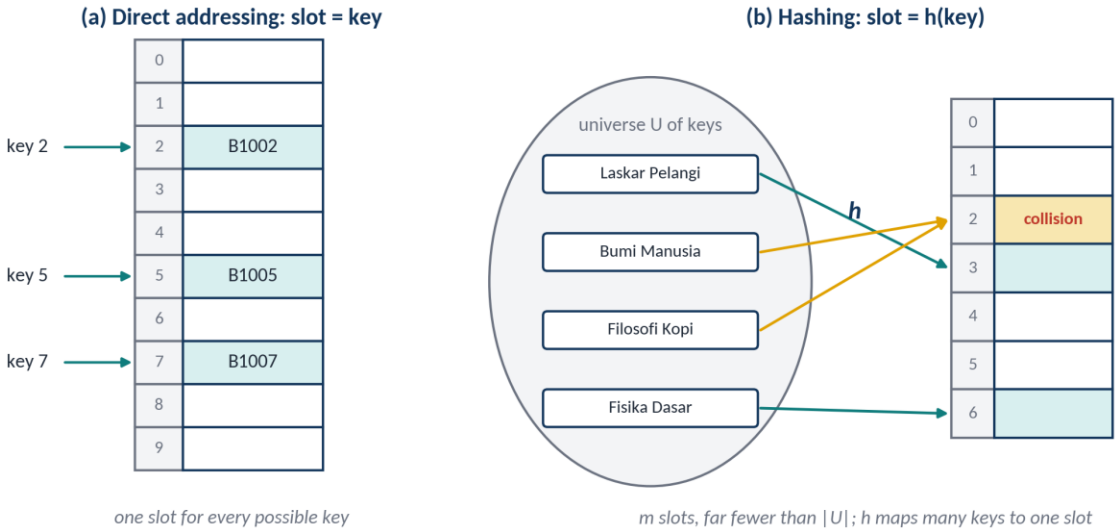


Figure 15.1 (a) A direct-address table has one slot for every possible key, so the key itself is the index. (b) A hash table has far fewer slots than possible keys; the hash function *h* maps each key to a slot, and two different keys may be sent to the same slot, a collision.

Figure 15.1*: (a) member numbers 0 to 999 index an array directly, O(1) in the worst case; but a table for all 64-bit keys would need 18 exabytes. (b) Hashing keeps the fast access and throws away the huge array. Book: Sections 15.2.2–15.2.3 and Figure 15.1.*

The Vocabulary

> **DEFINITION.** A *hash function* for a table of m slots is a function $h: U \to \{0,1,\ldots,m-1\}$; $h(k)$ is the *home slot* of k. A *hash table* is an array $T[0..m-1]$ in which k is stored at, or starting from, slot $h(k)$. Distinct keys with the same slot *collide*. The *load factor* of n keys in m slots is $\alpha = n/m$.

Luhn's IBM memorandum of 1953 already chained colliding records; the word *hashing* (chop and mix) appeared in print in 1968.

Hash tables sit inside every compiler, every database, every browser, and the core of Python, whose objects, modules and classes are dictionaries. Book: Section 15.2.3.

The Progression in One Table

Table 15.1 Searching for a key: from sequential search to hashing.

Structure	Search	Insert	Delete	Keeps order?	What it needs
Unsorted array (Section 2.3)	$\Theta(n)$	$O(1)$ at the end	$\Theta(n)$	no	nothing
Sorted array (Section 2.5)	$\Theta(\lg n)$	$\Theta(n)$	$\Theta(n)$	yes	keys that can be compared
Balanced search tree (Section 11.9)	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(\lg n)$	yes	keys that can be compared
Hash table (this chapter)	$O(1)$ average, $\Theta(n)$ worst	$O(1)$ average	$O(1)$ average	no	a good hash function and equality

IN PLAIN LANGUAGE. Searching by comparing is "higher or lower": about $\lg n$ questions. Hashing is knowing the house number: walk straight to the door.

Table 15.1*: the last column says what each structure costs besides time. Book: Section 15.2.4.*

What a Good Hash Function Must Do

Deterministic, **fast** (a few instructions per byte), **uniform** on the keys actually stored, and it **uses the whole key**.

> **DEFINITION.** Under *simple uniform hashing*, each key is equally likely to hash to any slot, independently of the others, Equation (15.1).

$$\Pr[h(k) = j] = \frac{1}{m} \quad \text{for every slot } j \in \{0,1,\dots,m-1\}. \tag{15.1}$$

No fixed function satisfies Equation (15.1) for every key set: an adversary who knows h can pick colliding keys. It is still the right target, and all the averages of this chapter are derived under it. Book: Section 15.3.1 and Equation (15.1).

Worked Example: the Division Method

$$h(k) = k \bmod m. \tag{15.2}$$

```
m = 10:  0  0  0  0  0  0  0  0
m = 16:  8  0  8  8  0  0  8  0
m = 11: 10  6  2  3  9  1  0  7
```

Eight book prices, all round thousands: $m = 10$ sees only the last digit; $m = 16$ only the last four bits; the prime $m = 11$ spreads them all.

Equation (15.2): avoid $m = 2^p$ and $m = 10^p$; prefer a prime not too close to a power of 2. A stride d with $\gcd(d, m) = g$ reaches only m/g slots. Book: Section 15.3.2 and Equation (15.2).

The Multiplication Method

$$h(k) = \lfloor m\,(kA \bmod 1) \rfloor, \tag{15.3}$$

$$h(k) = (k \cdot s \bmod 2^w) \gg (w - p), \qquad s = \lfloor 2^w A \rfloor. \tag{15.4}$$

Equation (15.3) works for any m, including powers of 2; Knuth recommends $A = (\sqrt{5} - 1)/2$. For $k = 45000$, $m = 16$: fractional part 0.529 ..., slot 8.

Equation (15.4) is Fibonacci hashing: one multiplication by `0x9E3779B97F4A7C15` and one shift, taking the top bits. The same eight prices spread over six of 16 slots. Book: Section 15.3.3.

Hashing Strings

Adding up character codes ignores order: *kasur* and *rusak* both sum to 550. Book codes B0000 to B1999 sum to 258–286: 72 of 101 buckets stay empty.

The cure: treat the string as a number in base B, the **polynomial hash** Equation (15.5):

$$h(s) = s_0 B^{L-1} + s_1 B^{L-2} + \cdots + s_{L-2} B + s_{L-1} = \sum_{i=0}^{L-1} s_i \, B^{L-1-i}. \tag{15.5}$$

Java's `String.hashCode` is Equation (15.5) with $B = 31$ in 32 bits; our programs use 64 bits. Now kasur gives 101820730 and rusak 108880570. Book: Section 15.3.4 and Equation (15.5).

Worked Example: Horner's Rule on "B1001"

Pseudocode 15.1 The polynomial hash of a string by Horner's rule; M is 2^{64} when the machine's unsigned overflow does the reduction for free.

```
POLY-HASH(s, L, B, M)
1  h = 0
2  for i = 0 to L - 1
3      h = (h * B + s[i]) mod M      // shift the old digits up, add the new one
4  return h
```

```
h = 0      * 31 + 66 = 66
h = 66     * 31 + 49 = 2095
h = 2095   * 31 + 48 = 64993
h = 64993  * 31 + 48 = 2014831
h = 2014831 * 31 + 49 = 62459810
```

Pseudocode 15.1*: one multiplication and one addition per character. 62459810 is exactly Java's* `"B1001".hashCode()`*. With unsigned 64-bit arithmetic the reduction modulo* 2^{64} *is free; Python writes* `& MASK`*. Book: Section 15.3.4.*

Hash Functions in C++

Listing 15.1 Fibonacci hashing, the poor sum of characters and the polynomial hash, from `hash_functions.cpp`.

```
// Multiplication method, integer form for m = 2^p ("Fibonacci hashing").
u64 hashMultInt(u64 k, int p) {
    const u64 s = 0x9E3779B97F4A7C15ULL;             // floor(2^64 * A)
    return (k * s) >> (64 - p);                        // the top p bits of k*s
}

// A poor string hash: add up the character codes.
u64 hashSum(const std::string &s) {
    u64 h = 0;
    for (unsigned char c : s) h += c;
    return h;
}

// Polynomial hash by Horner's rule, base 31, arithmetic modulo 2^64.
u64 hashPoly(const std::string &s) {
    u64 h = 0;
    for (unsigned char c : s) h = h * 31 + c;
    return h;
}
```

***Listing 15.1**: on 2000 codes and 101 buckets the polynomial hash uses every bucket and never puts more than 23 codes in one, near the ideal 19.8; the sum piles 150 into one. Book: Section 15.3.7.*

Rolling and Avalanche

$$h_i = (h_{i-1} - s_{i-1}B^{W-1}) \cdot B + s_{i+W-1}. \tag{15.6}$$

Equation (15.6): the next window's hash in $O(1)$, the heart of Rabin-Karp and content-defined chunking.

Avalanche: flipping one input bit should flip half the output bits. Identity 1.00, `k*31+7` 3.67, Fibonacci 16.02 but blind at bit 63, the SplitMix64 finalizer 31.97.

> **A HASH FOR TABLES IS NOT A HASH FOR PASSWORDS.** Table hashes aim at speed; password hashes (Argon2id, bcrypt) are deliberately slow and memory-hungry.

GCC's `std::hash<int>` is the identity; many modern tables finish every hash with a cheap mixing step. Book: Sections 15.3.5–15.3.7 and Equation (15.6).

The Birthday Problem

$$\Pr[\text{no collision among } n \text{ keys}] = \prod_{i=1}^{n-1}\left(1 - \frac{i}{m}\right). \tag{15.7}$$

$$\Pr[\text{no collision}] \approx \exp\left(-\sum_{i=1}^{n-1}\frac{i}{m}\right) = \exp\left(-\frac{n(n-1)}{2m}\right). \tag{15.8}$$

Collisions cannot be avoided (pigeonhole). The surprise is how *soon*: 23 people give even odds of a shared birthday.

Equation (15.7): the $(i+1)$-st key avoids i occupied slots with probability $1 - i/m$. With $1 - x \approx e^{-x}$ the product becomes Equation (15.8). Book: Sections 15.4.1–15.4.2.

The Square Root Is the Whole Story

$$n_{1/2} \approx \sqrt{2\ln 2 \cdot m} \approx 1.1774\sqrt{m}. \tag{15.9}$$

$$E[\text{number of colliding pairs}] = \frac{n(n-1)}{2m}. \tag{15.10}$$

Table 15.2 How many random keys make a collision likely, computed from Equation (15.7); the middle column is the approximation Equation (15.9).

Slots m	n For a 50% chance	$1.1774\sqrt{(m)}$	n For a 99% chance
365	23	22.5	57
1000	38	37.2	95
10^6	1178	1177.4	3034
2^{32}	77,164	77,162.7	198,892
2^{64}	—	5.057×10^9	1.303×10^{10}

***Table 15.2**: even 2^{64} hash values collide after about five billion keys. By Equation (15.10), 23 keys in 365 slots expect 0.69 colliding pairs. Book: Section 15.4.3.*

The Birthday Curve

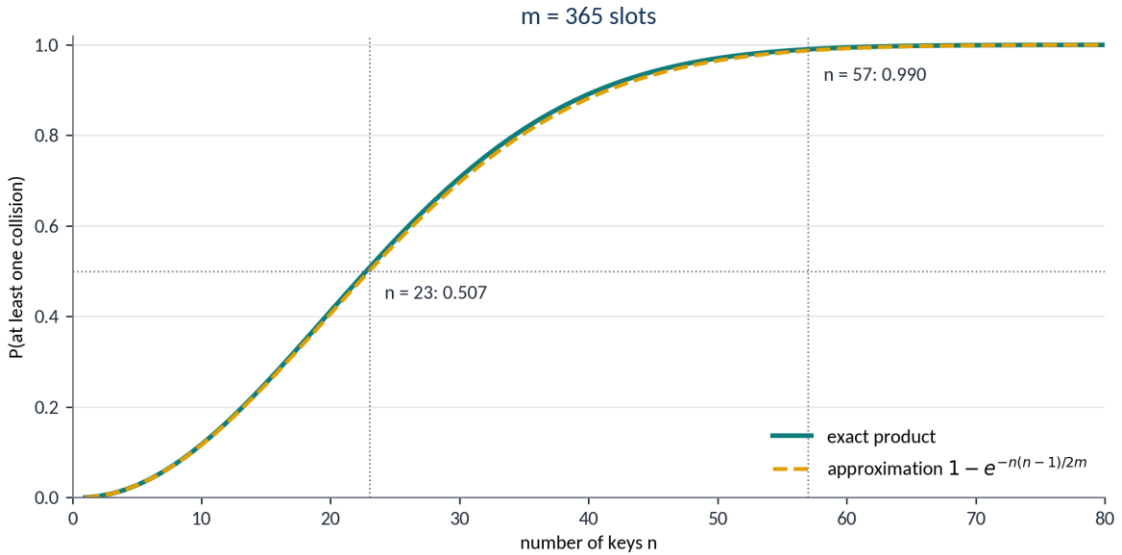


Figure 15.2 The probability of at least one collision among n random keys in 365 slots: 0.507 at $n = 23$ and 0.990 at $n = 57$. The dashed curve is the approximation Equation (15.8).

Figure 15.2*: a table cannot avoid collisions, since it would need about n^2 slots; it plans for them. 1000 keys in 1000 buckets leave 353 empty. Book: Sections 15.4.3–15.4.4 and Figure 15.2.*

Separate Chaining

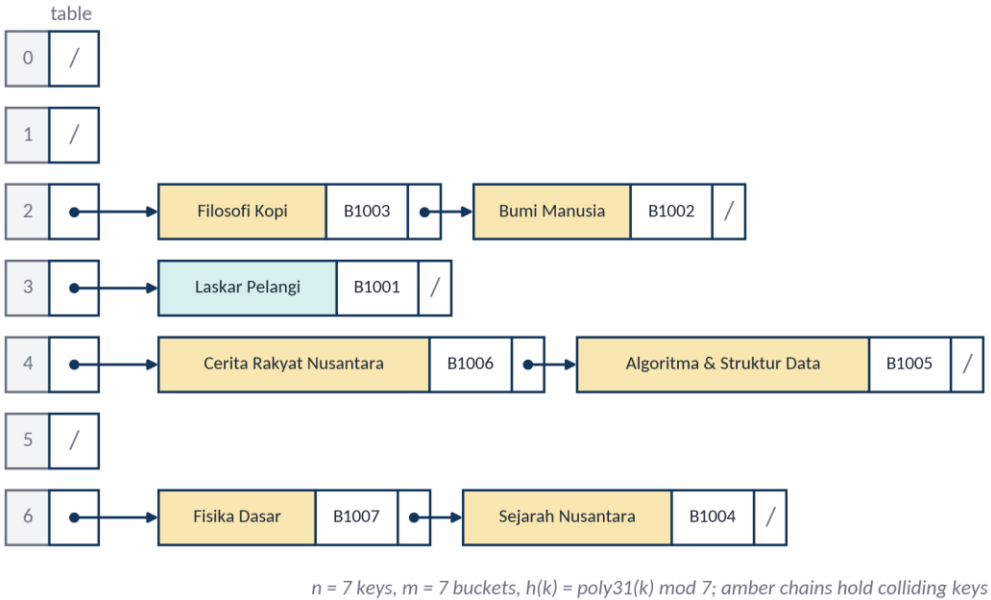


Figure 15.3 Separate chaining: seven titles in seven buckets with $h(k) = \text{poly31}(k) \bmod 7$. Each bucket holds the head of a singly linked list; buckets 2, 4 and 6 hold two colliding titles each, and buckets 0, 1 and 5 are empty.

***Figure 15.3**: every slot holds the head of a singly linked list of Chapter 7; buckets 2, 4 and 6 hold two titles each. Book: Section 15.5.1 and Figure 15.3.*

Search, Insert, Delete with Chains

Pseudocode 15.2 Search, insertion and deletion in a hash table with separate chaining.

```
CHAINED-SEARCH(T, k)
1  p = T[h(k)]                      // head of the chain for k
2  while p != NIL and p.key != k
3      p = p.next
4  return p                         // NIL if k is absent

CHAINED-PUT(T, k, v)
1  p = CHAINED-SEARCH(T, k)
2  if p != NIL
3      p.value = v                  // present: update in place
4  else
5      insert a new node (k, v) at the front of chain T[h(k)]
6      n = n + 1
7      if n / m > maxLoad
8          REHASH(T, next prime >= 2m)

CHAINED-DELETE(T, k)
1  walk chain T[h(k)] with prev and cur until cur.key == k
2  unlink cur (as DELETE-KEY of Section 7.6.5) and free it
```

Pseudocode 15.2*: insert at the front after a search (an existing key is updated, not duplicated); delete is* `deleteKey` *of Section 7.6.5 with its trailing pointer; line 7 watches the load factor. Book: Section 15.5.1.*

`put` in C++

Listing 15.2 `put` from `chaining.cpp`: update in place, or insert at the front and rehash when the load passes the maximum.

```
    // Insert key with value, or replace the value if key is already present.
    void put(const std::string &key, const std::string &value) {
        std::size_t b = bucketOf(key);
        for (Node *p = table_[b]; p != nullptr; p = p->next) {
            if (p->key == key) {                    // present: update in place
                p->value = value;
                return;
            }
        }
        table_[b] = new Node{key, value, table_[b]};   // absent: front insertion
        ++n_;
        if ((double)n_ / table_.size() > maxLoad_)
            rehash(nextPrime(2 * table_.size()));
    }
```

Listing 15.2*: the eighth title makes $8/7 > 1$; the table grows to 17 buckets, the next prime after 14, moving nodes rather than copying them. Valgrind: all 25 blocks freed. Book: Sections 15.5.2–15.5.3.*

How Long Are the Chains?

$$E[\text{comparisons, unsuccessful}] = \alpha, \qquad \text{time } \Theta(1+\alpha). \tag{15.11}$$

$$E[\text{comparisons, successful}] = \frac{1}{n}\sum_{i=1}^{n}\left(1+\frac{n-i}{m}\right) = 1+\frac{n-1}{2m} \approx 1+\frac{\alpha}{2}. \tag{15.12}$$

A key lands in a given bucket with probability $1/m$: the expected chain length is α.

Keep α below a constant by growing m with n, and every operation is $O(1)$ on average.

Equation (15.11) walks a whole chain; Equation (15.12) passes the keys inserted after x into its bucket. Both depend on α only, not on n. Book: Section 15.5.4.

Open Addressing

Every key lives *inside* the array, one per slot; no pointers, no nodes, one cache-friendly block, but $\alpha \leq 1$.

$$h(k,0),\ h(k,1),\ \ldots,\ h(k,m-1), \tag{15.13}$$

Insertion walks Equation (15.13) to an empty slot; search walks the *same* sequence until the key or an empty slot, which proves absence.

The running trace inserts 1003, 1019, 1035, 1012, 1028, 1013, 1044, 1051 into 16 slots: four want slot 11, three want slot 4. Book: Section 15.6.1 and Equation (15.13).

Three Probe Sequences

$$h(k,i) = (h_1(k) + i) \bmod m. \tag{15.14}$$

$$h(k,i) = \left(h_1(k) + \frac{i(i+1)}{2}\right) \bmod m. \tag{15.15}$$

$$h(k,i) = (h_1(k) + i \cdot h_2(k)) \bmod m. \tag{15.16}$$

Equation (15.14) suffers primary clustering: runs merge and grow. Equation (15.15) with triangular offsets visits every slot of a power-of-two table, but keys with the same home follow the same path. Equation (15.16) lets the step depend on the key. Book: Sections 15.6.2–15.6.4.

Worked Example: Eight Keys, Three Ways

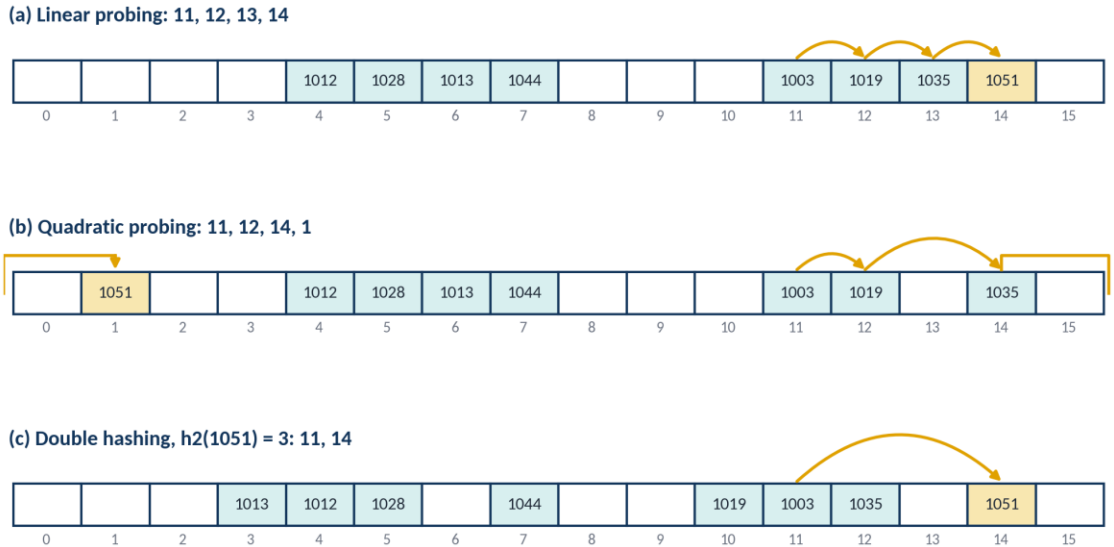


Figure 15.4 The same eight keys in 16 slots with (a) linear probing, (b) quadratic probing and (c) double hashing. The arrows are the probes made to insert (and later to find) 1051: four, four and two.

Figure 15.4*: inserting 1051 takes 4, 4 and 2 probes; all eight keys take 19, 18 and 15. Book: Section 15.6 and Figure 15.4.*

Deletion Needs Tombstones

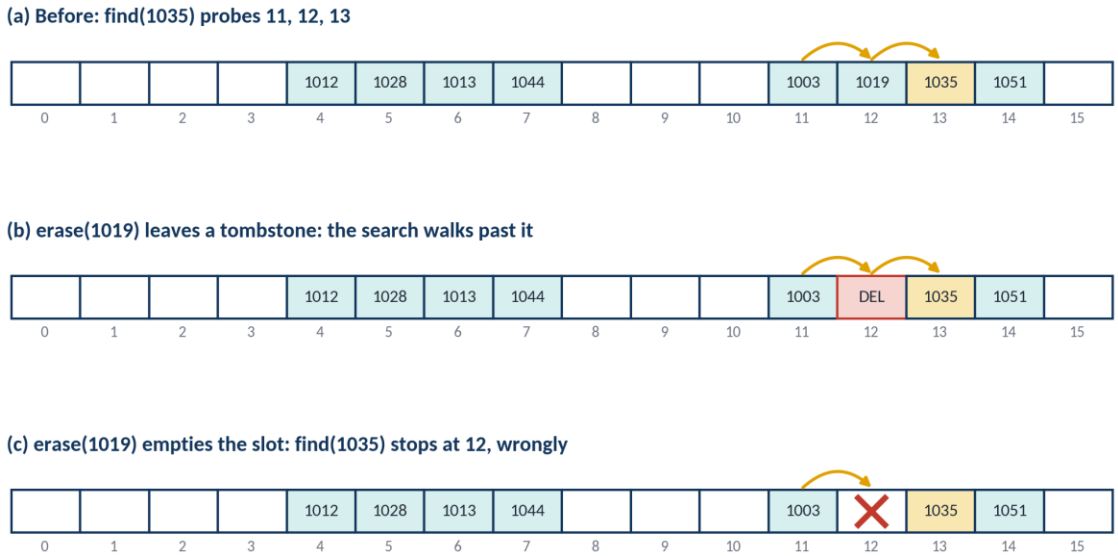


Figure 15.5 Deleting 1019 from the linear-probing table. (a) Before, a search for 1035 probes 11, 12, 13. (b) With a tombstone in slot 12 the search walks past it and still finds 1035. (c) If slot 12 is simply emptied, the search stops there and wrongly reports 1035 absent.

Figure 15.5*: emptying slot 12 cuts the probe path of 1035, which is then "not found". A DELETED marker is walked past by search and reused by insertion. Book: Section 15.6.5 and Figure 15.5.*

`find` and `erase` in C++

Listing 15.3 `find` and `erase` from `open_addressing.cpp`: EMPTY ends the search, DELETED does not.

```
    // Return the slot holding k, or -1; `probes` receives the probes used.
    int find(long long k, int &probes) const {
        probes = 0;
        for (int i = 0; i < m_; ++i) {
            int s = slot(k, i);
            probes = i + 1;
            if (state_[s] == EMPTY) return -1;       // the probe sequence ends here
            if (state_[s] == OCCUPIED && key_[s] == k) return s;
        }                                            // DELETED: keep going
        return -1;
    }

    bool erase(long long k) {                        // leave a tombstone
        int probes, s = find(k, probes);
        if (s < 0) return false;
        state_[s] = DELETED;
        return true;
    }
```

***Listing 15.3**. Insertion must walk on past a tombstone to an EMPTY slot before reusing the first tombstone, or it could store a second copy of the key. Book: Sections 15.6.5–15.6.6.*

Chaining or Open Addressing?

Table 15.3 Separate chaining and open addressing compared.

	Separate chaining	Open addressing
Where colliding keys live	in list nodes outside the array	in other slots of the array
Load factor	any α; performance degrades gently	$\alpha < 1$ strictly; keep it at most about 0.7–0.9
Memory per key	key, value and one pointer, plus allocator overhead	key and value only, plus empty slots
Cache behavior	a pointer jump per node	consecutive slots (linear) or scattered (double)
Deletion	unlink the node, as in Chapter 7	tombstones or backward shift
Pointers to stored elements	stay valid when the table grows	move when the table grows

***Table 15.3**: C++ promises that references survive a rehash and exposes a bucket interface, which only chaining makes natural. Book: Section 15.6.7.*

The One Number That Matters

$$\alpha = \frac{n}{m}, \tag{15.17}$$

$$E[\text{probes, unsuccessful}] \leq 1 + \alpha + \alpha^2 + \cdots = \frac{1}{1-\alpha}. \tag{15.18}$$

$$E[\text{probes, successful}] \leq \frac{1}{n}\sum_{i=0}^{n-1}\frac{1}{1-i/m} \approx \frac{1}{\alpha}\int_0^{\alpha}\frac{dx}{1-x} = \frac{1}{\alpha}\ln\frac{1}{1-\alpha}. \tag{15.19}$$

Equation (15.17): the promise of $O(1)$ is a promise that α stays bounded, and the table must keep it. Under uniform hashing, Equation (15.18) and Equation (15.19); double hashing approximates it. Book: Sections 15.7.1–15.7.2.

Linear Probing, and the Measurements

$$E[\text{probes, unsuccessful}] \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right), \tag{15.20}$$

$$E[\text{probes, successful}] \approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right). \tag{15.21}$$

	chaining		linear		quadratic		double	
alpha	succ	unsucc	succ	unsucc	succ	unsucc	succ	unsucc
0.50	1.25	0.50	1.51	2.52	1.43	2.15	1.39	2.03
0.75	1.38	0.76	2.50	8.81	2.01	4.56	1.86	4.01
0.90	1.46	0.90	5.60	50.07	2.81	11.74	2.56	9.87

Equation (15.20) and Equation (15.21) are Knuth's first analysis of an algorithm (1962): the square is primary clustering. At $\alpha = 0.9$: 50.5 predicted, 50.07 measured. Book: Sections 15.7.2–15.7.3.

Measured against Predicted

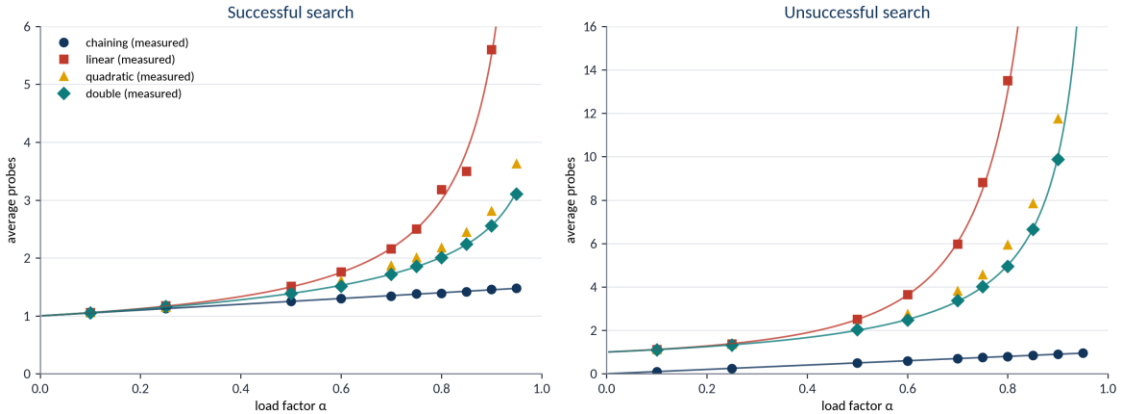


Figure 15.6 Average probes per search in a table of 16,384 slots, measured (markers) and predicted by Equation (15.12), Equation (15.11), Equation (15.20), Equation (15.19) and Equation (15.18) (curves). Chaining stays flat; linear probing explodes first; double hashing follows the uniform-hashing curve.

Figure 15.6*: the theory is right; near-full linear probing is erratic; chaining degrades gracefully. Libraries rehash at 1.0 (C++), 0.75 (Java), 2/3 (Python). Book: Section 15.7.3 and Figure 15.6.*

How Chain Lengths Are Distributed

$$\Pr[\text{a bucket holds exactly } j \text{ keys}] \approx \frac{e^{-\alpha}\alpha^{j}}{j!}. \tag{15.22}$$

For $\alpha = 1$: 36.8% of buckets empty, 36.8% with one key, 18.4% with two, 6.1% with three.

Measured on $n = m = 16384$: 6108 empty (6027 predicted), longest chain 7.

Equation (15.22) explains why Java's `HashMap` turns a bin into a tree only at 8 entries: with random hash codes that happens about once in ten million. Book: Section 15.7.4 and Equation (15.22).

Chain Lengths, Plotted

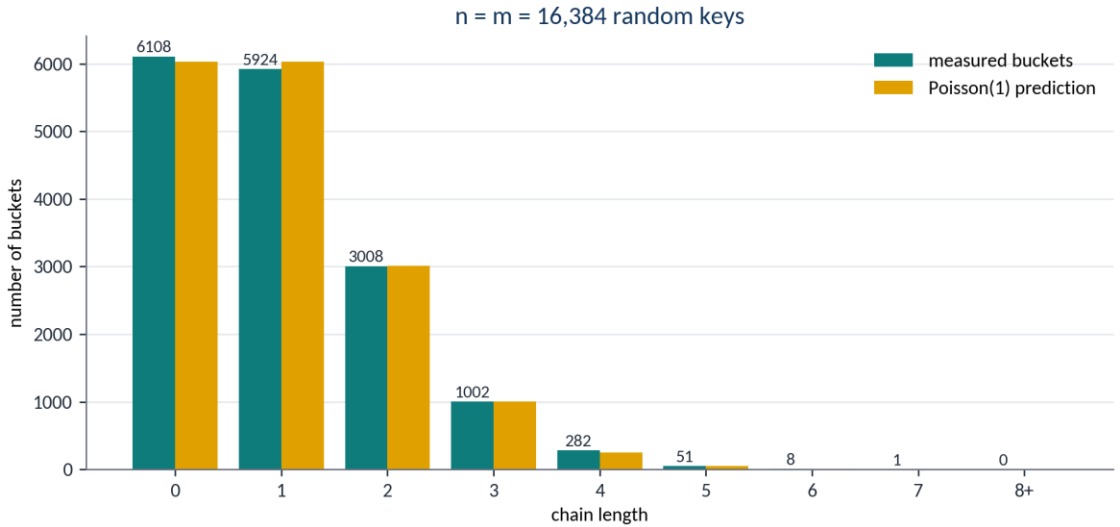


Figure 15.7 Chain lengths for 16,384 random keys in 16,384 buckets (teal) against the Poisson prediction Equation (15.22) with $\alpha = 1$ (amber). More than a third of the buckets are empty; none holds more than 7 keys.

***Figure 15.7**: more than a third of the buckets are empty; none holds more than 7 keys. Book: Section 15.7.4 and Figure 15.7.*

Rehashing, and Why Doubling Is Cheap

Every slot depends on m, so a larger table means reinserting every key: $\Theta(n)$ for one rehash.

Doubling makes the moves add up to less than $2n$, Equation (15.23):

$$\text{total keys moved} \leq n\left(1 + \frac{1}{2} + \frac{1}{4} + \cdots\right) < 2n. \tag{15.23}$$

Measured: 200,000 insertions, 16 rehashes, 393,226 moves, 1.966 per insertion.

Growing by a constant amount instead is $\Theta(n^2)$ (Exercise 15.13). Call `reserve(n)` or give Java a capacity when the count is known in advance. Book: Section 15.7.5 and Equation (15.23).

The Complete Bill

Table 15.4 The cost of hash-table operations with the load factor Equation (15.17) held below a constant; compare the balanced search tree's $\Theta(\lg n)$ for all of them.

Operation	Chaining, average	Open addressing, average	Worst case (both)
search, successful	$1 + \alpha/2$ comparisons	$\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$ probes (uniform)	$\Theta(n)$
search, unsuccessful	α comparisons	$\frac{1}{1-\alpha}$ probes (uniform)	$\Theta(n)$
insert (with duplicate check)	as unsuccessful search, then $O(1)$	as unsuccessful search	$\Theta(n)$
delete	as successful search, then $O(1)$	as successful search	$\Theta(n)$
insert, including rehashing	$O(1)$ amortized	$O(1)$ amortized	$\Theta(n)$ for one insertion
list all keys in sorted order	$\Theta(n\lg n)$ (sort first)	$\Theta(n\lg n)$	$\Theta(n\lg n)$
minimum, successor, range query	$\Theta(n+m)$	$\Theta(m)$	$\Theta(n+m)$

***Table 15.4**: the worst case is every key in one slot: one long list or one long run, $n(n-1)/2$ comparisons to insert n keys. Book: Section 15.8.1.*

Hash Flooding

An attacker who knows h sends keys that all collide: Crosby and Wallach (2003); Klink and Wälde (2011) slowed almost every web platform with one crafted request.

"Aa" and "BB" collide under base 31; 11 blocks give $2^{11} = 2048$ colliding 22-letter strings.

```
2. n = 2048 keys into m = 4096 buckets
  keys and hash function        longest   insert cmp   search cmp
  ordinary keys, hash31               3          935         2983
  attack keys, hash31              2048      2096128      2098176
  attack keys, keyed hash             4          429         2477
```

2,096,128 comparisons, exactly $2048 \cdot 2047/2$*, against 935 for ordinary keys; with a million keys the table would take hours to fill. Book: Section 15.8.2.*

The Defense: Randomized Hashing

DEFINITION. A family $\mathcal{H}$ of hash functions is *universal* if for every pair of distinct keys, Equation (15.24) holds over the random choice of h.

$$\Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq \frac{1}{m}, \tag{15.24}$$

$$h_r(s) = \left(\sum_{i=0}^{L-1} s_i \; r^{L-1-i} \right) \bmod p. \tag{15.25}$$

Equation (15.25) with a secret r modulo $2^{31} - 1$: two different strings collide with probability at most $(L-1)/p$; the attack keys scatter (longest chain 4). Python uses SipHash-1-3 keyed at startup; Java 8 treeifies crowded bins. Book: Section 15.8.3.

`std::map` versus `std::unordered_map`

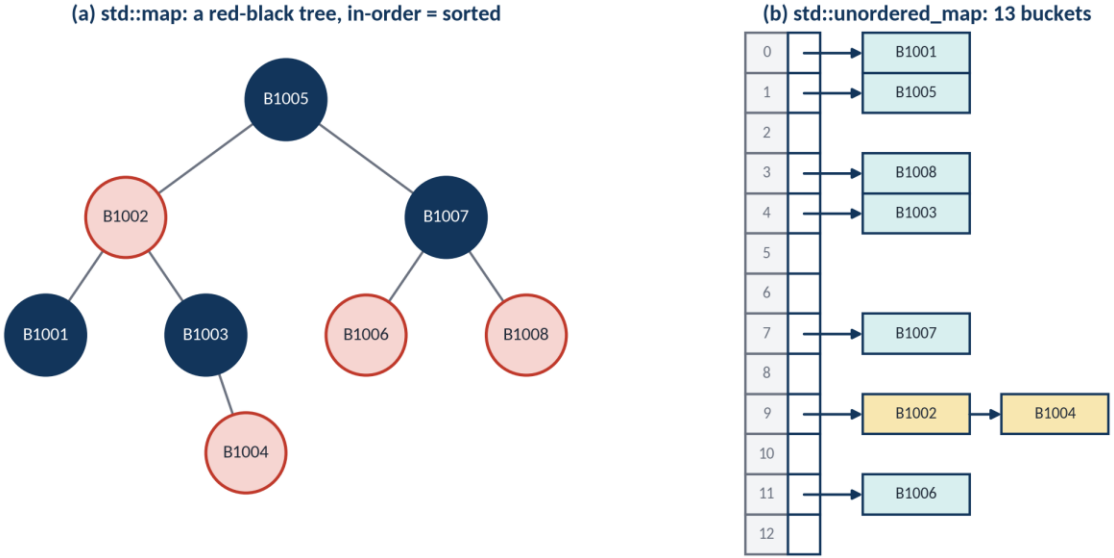


Figure 15.8 The eight codes inserted in the order B1005, B1001, B1008, B1003, B1002, B1007, B1004, B1006. (a) `std::map` keeps a red-black tree, whose in-order traversal is sorted. (b) `std::unordered_map` keeps 13 buckets; B1002 and B1004 share bucket 9, and no order survives.

***Figure 15.8**: the tree returns the codes sorted; the hash table in an order that is neither sorted nor insertion order, and may change after the next rehash. Book: Section 15.9.1 and Figure 15.8.*

Same Interface, Different Machines

Table 15.5 `std::map` and `std::unordered_map` compared.

	`std::map`	`std::unordered_map`
Structure	red-black tree (Section 11.9.3)	array of buckets with chains
Key needs	`operator<`	`std::hash<Key>` and `operator==`
Iteration order	sorted by key	unspecified, may change on rehash
Search, insert, erase	$O(\lg n)$ worst case	$O(1)$ average, $O(n)$ worst case
`lower_bound`, range queries, min, max	yes, $O(\lg n)$	no
Memory per element	node with three pointers and a color	node with one pointer, plus the bucket array
Use it when	order, ranges or guaranteed bounds matter	only exact-key lookups matter

***Table 15.5**: `std::map` is a red-black tree, whatever the problem calls it. On a million random keys the hash table inserted about 4 times and found about 25 times faster: the tree pays a cache miss per level. Book: Section 15.9.*

Inside Java's `HashMap`

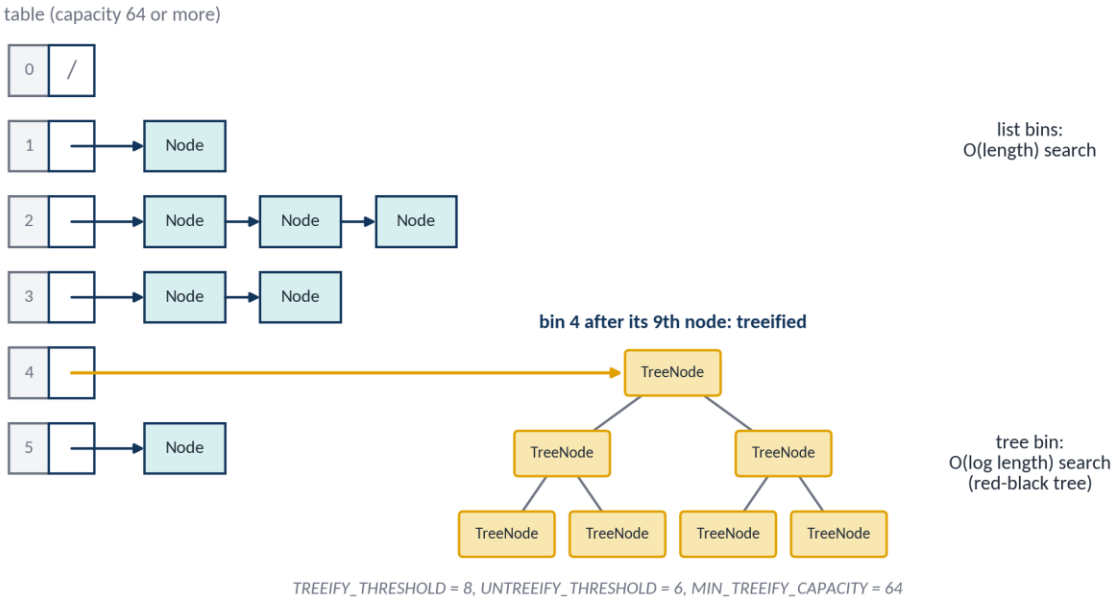


Figure 15.9 A `HashMap` table with list bins and one treeified bin. A bin becomes a red-black tree of `TreeNode`s when it collects 8 entries in a table of at least 64 bins, so a search inside it takes $O(\lg \ell)$ instead of $O(\ell)$ steps for a bin of ℓ entries.

***Figure 15.9**: power-of-two tables from 16, load 0.75, `h ^ (h >>> 16)` then `(n - 1) & hash`; 65,536 colliding keys cost only about 13 times ordinary keys in a treeified bin. Book: Section 15.10.1 and Figure 15.9.*

Inside Python's `dict`

$$j \leftarrow (5j + 1 + \text{perturb}) \bmod 2^k, \qquad \text{perturb} \leftarrow \text{perturb} \gg 5, \tag{15.26}$$

Open addressing in 2^k slots; *perturb* shifts the high bits in, like double hashing, and then $5j + 1$ visits every slot.

Compact since 3.6: a small `indices` table points into a dense `entries` array, so iteration follows insertion order (guaranteed since 3.7). At most 2/3 full.

Equation (15.26). Integers hash to themselves modulo $2^{61} - 1$ (`hash(-1)` is -2); strings use SipHash-1-3 with a per-process key. Book: Section 15.10.2 and Equation (15.26).

The Compact Layout

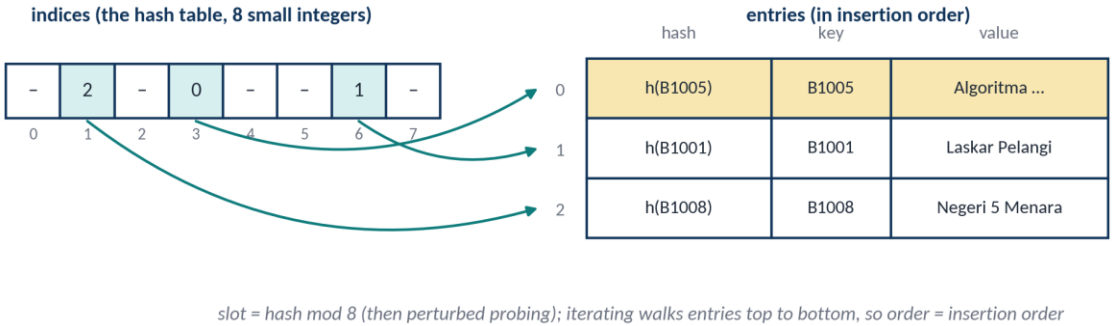


Figure 15.10 The compact `dict` layout with three items: a small `indices` table of 8 one-byte slots is probed by hash, and each used slot holds the position of an entry in the dense `entries` array, which is kept in insertion order. (The slot positions shown are illustrative; real hash values change from run to run.)

***Figure 15.10**: probing happens in the small `indices` array; the large entries never move for a collision. A `set` has no `entries` array, so its order is not insertion order. Book: Section 15.10.2 and Figure 15.10.*

Three Libraries Side by Side

Table 15.6 The hash tables of the C++, Java and Python standard libraries (GCC 13, JDK 25, CPython 3.14).

	C++ `std::unordered_map`	Java `HashMap`	Python `dict`
Collision handling	separate chaining	separate chaining, bins become red-black trees at 8	open addressing, perturbed probing Equation (15.26)
Table size	prime (GCC)	power of 2, from 16	power of 2, from 8
Maximum load	1.0 (adjustable)	0.75 (adjustable)	2/3
Default hash of an integer	identity (GCC)	`Integer.hashCode` = the value, then spread	the value modulo $2^{61} - 1$
Default hash of a string	fixed-seed byte hash	$\sum s_i 31^{L-1-i}$ in 32 bits, cached	SipHash-1-3, random key per process
Iteration order	unspecified	unspecified (`LinkedHashMap`: insertion)	insertion order
Ordered sibling	`std::map`	`TreeMap`	none built in (sort, or `bisect`)

Table 15.6 *gathers Sections 15.9 and 15.10 (GCC 13, JDK 25, CPython 3.14). Book: Section 15.10.*

Applications, and an LRU Cache

Listing 15.4 The class `LruCache` from `applications.cpp`: a `std::list` in recency order and an `unordered_map` from key to list iterator.

```
class LruCache {                                    // a hash map + a linked list
public:
    explicit LruCache(std::size_t capacity) : capacity_(capacity) {}

    bool get(const std::string &key) {              // true on a hit
        auto it = pos_.find(key);
        if (it != pos_.end()) {                     // hit: move to the front
            order_.splice(order_.begin(), order_, it->second);
            return true;
        }
        if (order_.size() == capacity_) {           // full: evict the back
            pos_.erase(order_.back());
            order_.pop_back();
        }
        order_.push_front(key);                     // miss: load, put in front
        pos_[key] = order_.begin();
        return false;
    }
```

***Listing 15.4**: ten requests, five hits. The same program counts words (`++count[w]`), detects duplicate returns, and finds two prices summing to Rp119,000 in one pass. Reverse maps serve Chapter 16's itinerary. Book: Section 15.11.*

Profitina in Practice

One page, many addresses: canonicalize URLs, then a set of visited pages answers "seen before?" in $O(1)$.

Near-duplicate pages: rolling-hash fingerprints of word windows, compared with set lookups.

Screening links, **counting per user**, **caching expensive answers**: set membership, counters keyed by account, an LRU cache.

The other kind of hash: passwords are stored with Argon2id, the opposite of a fast table hash.

Profitina is an AI visibility (GEO) platform built with FastAPI on PostgreSQL; we describe the shapes, not the product's implementation. Book: Section 15.12.

Common Mistakes

Remainder by a power of two with a weak hash; **adding up characters**; **negative remainders** (`-7 % 5` is −2 in C++ and Java).

Changing a key while stored; `equals` **without the matching** `hashCode`.

Emptying a slot on deletion; **stopping insertion at the first tombstone**; **letting the load factor grow**.

Relying on iteration order; **calling** `std::map` **a hash table**; `m[key] == 0` **to test presence**; **public hashes for strangers' keys**.

And never a table hash for passwords, or a password hash for tables. Book: Section 15.13.

Summary: Five Things to Remember

1. Hashing computes the slot from the key and escapes the comparison lower bound: $O(1)$ average, no order, no guarantee.

2. Good functions: division with a prime m, Fibonacci multiplication taking top bits, the polynomial string hash by Horner.

3. Collisions come at about $\sqrt{m}$ keys; chaining costs about $1 + \alpha/2$ and α, open addressing $1/(1 - \alpha)$, and needs tombstones.

4. Keep α bounded by doubling: under two moves per insertion.

5. The worst case is $\Theta(n)$ and can be provoked; keyed hashing and treeified bins defend. C++ chains, Java chains into trees, Python probes a compact index.

Book: Section 15.14.

Next Time

Chapter 16 — Evaluation and Discussion IV. Graphs, heaps and hash tables at work, and problems that draw on the whole book.

The itinerary from a pile of tickets returns, with a map and a reverse map.

Before then: work Exercises 15.1, 15.2, 15.7 and 15.12, in C++ and then in Java or Python.

The solutions, with programs, are in the book. Book: Sections 15.15 and 15.16.

Lecture 16 — Evaluation and Discussion IV: Graphs, Heaps, and Hash Tables at Work

Course: Data Structure (EF234201)

Book: Irfan Subakti, *Data Structure with C++, Java and Python*, Chapter 16

Lecturer: Irfan Subakti — yifana@gmail.com

No new structure today: nine problems that put the graph, the heap and the hash table to work, and the question the whole book prepares for, which structure to choose. Book: Chapter 16, Sections 16.1–16.12.

What This Meeting Covers

Table 16.1 The nine problems of this chapter, the section that solves each one, and the chapters that supply its main idea.

Problem	Section	What it asks	Main tool from
16.1	16.3	Complete a max-heap class: `extractMax`, `increaseKey`, `getMin`, `buildMaxHeap`, `isMaxHeap`	Ch. 3, 11, 14
16.2	16.4	Put a pile of travel tickets in order, with a hash map	Ch. 7, 15
16.3	16.5	The fewest hops between library branches	Ch. 5, 13
16.4	16.5	An order in which to read topics that depend on each other	Ch. 5, 13
16.5	16.5	The quickest route for the book van	Ch. 13, 14
16.6	16.6	The k largest values of a long list, with a heap of size k	Ch. 14
16.7	16.6	The median so far, with two heaps	Ch. 14
16.8	16.7	A hash table with separate chaining, built by hand	Ch. 7, 15
16.9	16.7	Open addressing, and why deletion needs a tombstone	Ch. 15

Table 16.1: for every problem we draw the structure, trace it by hand, read the code, and run a tempting wrong solution. Then Section 16.8 measures every structure of the book on the same ten thousand books. Book: Section 16.1.

Where We Stand: One Idea

Chapter 13: graphs, adjacency lists, BFS with a queue, DFS with a stack, topological sort, Dijkstra. **Chapter 14**: the binary heap in an array, sift-up and sift-down in $O(\lg n)$. **Chapter 15**: hash functions, chaining, open addressing.

> **THE ONE IDEA BEHIND ALMOST EVERY PROBLEM HERE.** *Every structure promises something, an invariant, and every operation must leave the promise true.* A heap: no child larger than its parent. BFS: vertices leave the queue in order of distance. Dijkstra: the vertex taken from the heap has its final distance. A hash table: a key is in the slot its hash names, or further along the search path.

When a program in this chapter goes wrong, it is almost always because an operation left the promise broken. Book: Section 16.1.

The Nine-Step Method

Table 16.2 The nine-step method for a problem about a data structure. Steps 1 to 3 understand the problem, Steps 4 to 6 design the operations, Step 7 checks that the design is worth having, and Steps 8 and 9 check the program.

Step	Write on paper	Question it answers
1	The problem in your own words, and one example by hand	Do I know what the output must be?
2	The structure, and one sentence of reason	Heap, hash table, graph, tree, list or array?
3	The picture: tree and array, circles and lists, slots and chains	What does the structure look like right now?
4	The invariant	What does the structure promise?
5	The edge cases	Empty, one element, absent key, collision, cycle?
6	The changes, numbered	Is anything lost or seen too late in this order?
7	The cost, and the cost of the slow way	Is the structure worth it?
8	What to do with input that breaks the rules	Message, exception, or special value?
9	Tests, a memory checker, and a slow twin	Does it work, and does it only *seem* to work?

Table 16.2: the seven steps of Chapter 8 plus two new ones: Step 7 counts the cost, Step 8 decides what to do with input that breaks the rules. Book: Section 16.2.

Three Structures, Three Languages

Table 16.3 The structures of Chapters 13 to 15 in the three languages (a selection of the book's table). In C++ the word `map` means a search tree, not a hash table.

Idea	C++	Java	Python
Priority queue	`std::priority_queue<T>`	`PriorityQueue<T>`	`heapq` on a `list`
Which end comes out first	the **largest**	the **smallest**	the **smallest**
Hash table (map)	`std::unordered_map<K, V>`	`HashMap<K, V>`	`dict`
Ordered map (a balanced search tree)	`std::map<K, V>`	`TreeMap<K, V>`	none
FIFO queue for BFS	`std::queue<int>`	`ArrayDeque<Integer>`	`collections.deque`

Table 16.3: two rows cause most bugs. `std::priority_queue` hands out the largest, `PriorityQueue` and `heapq` the smallest; and `std::map` is a search tree, the hash table is `std::unordered_map`. Book: Section 16.2.

Problem 16.1 — Completing a Max-Heap Class

PROBLEM. Add to the class `MaxHeap` (a `vector<int> heap` with `heapifyDown` and `heapifyUp`): (1) `extractMax()`; (2) `increaseKey(index, newValue)`; (3) `getMin()`; (4) `buildMaxHeap(arr)`; (5) `isMaxHeap(arr)`; and `clear()`. Insert 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, then test each.

$$parent(i)=\left\lfloor \frac{i-1}{2} \right\rfloor, left(i)=2i+1, right(i)=2i+2. \tag{16.1}$$

DEFINITION. The *max-heap invariant*: `heap[parent(i)] >= heap[i]` for every $1 \le i < n$. So `heap[0]` is the largest element.

Equation (16.1): the relatives of index i are found by arithmetic. Each new operation is a one-sentence job read off the invariant. Book: Sections 16.3.1–16.3.2.

The Heap After Eleven Insertions

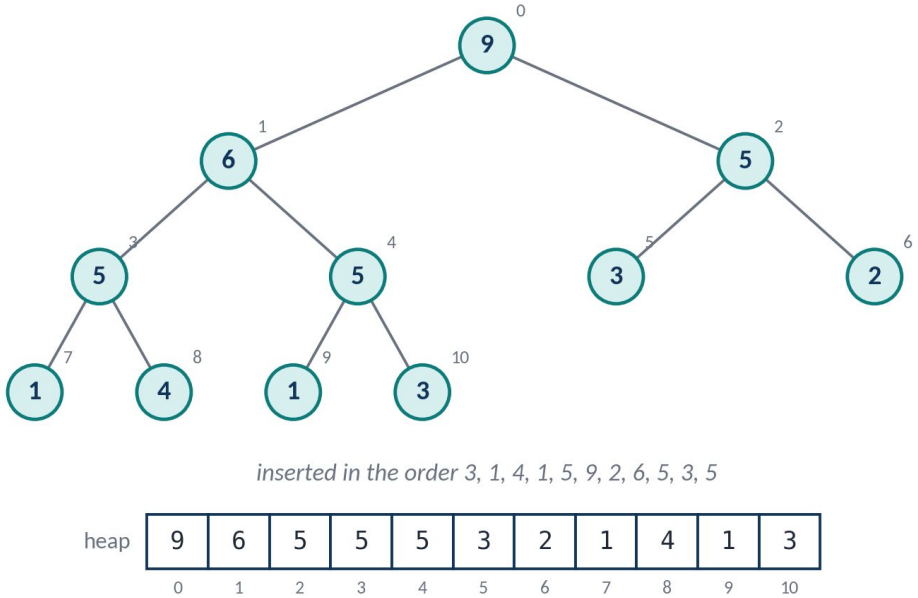


Figure 16.1 The max-heap after the eleven insertions of the driver, drawn as a tree (small grey numbers are array indices) and as the array it really is.

Figure 16.1: every parent is at least its children, yet the array is not sorted, and need not be. Book: Section 16.3.3 and Figure 16.1.

Worked Example: `extractMax` in Three Moves

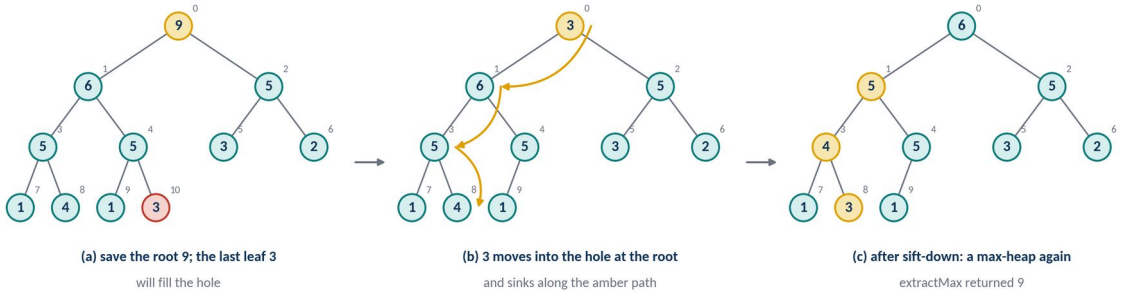


Figure 16.2 `extractMax` in three moves: save the root, move the last leaf into the root, and let it sink along the path of larger children until the max-heap invariant holds again.

Figure 16.2: save 9, move the last leaf 3 to the root, sink it past 6, 5 and 4: `[6, 5, 5, 4, 5, 3, 2, 1, 3, 1]`*. Book: Section 16.3.3 and Figure 16.2.*

`extractMax` and `increaseKey` in C++

Listing 16.1 `extractMax` and `increaseKey` from `p1601_max_heap.cpp`: save the answer first; a larger key can only move up.

```
int extractMax() {
    if (heap.empty()) throw underflow_error("extractMax on an empty heap");
    int maxVal = heap[0];            // SAVE the answer first
    heap[0] = heap.back();           // the last leaf fills the hole at the root
    heap.pop_back();
    if (!heap.empty()) heapifyDown(0);
    return maxVal;
}

void increaseKey(int index, int newValue) {
    if (index < 0 || static_cast<size_t>(index) >= heap.size())
        throw out_of_range("increaseKey: no element at that index");
    if (newValue < heap[index])
        throw invalid_argument("increaseKey: new value is smaller");
    heap[index] = newValue;
    heapifyUp(index);
}
```

$$h = \lfloor \lg n \rfloor . \tag{16.2}$$

Listing 16.1: both follow one path, so both cost $O(\lg n)$*, since* $2^h \le n$ *gives Equation (16.2). A decrease is refused; Exercise 16.1 writes* `decreaseKey`*. Book: Sections 16.3.3–16.3.4.*

`increaseKey(2, 8)`: One Swap Up

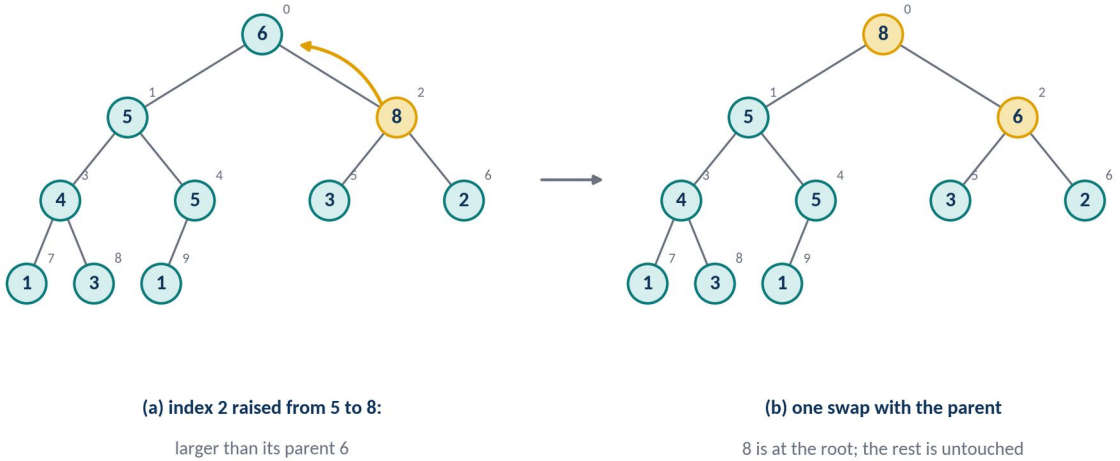


Figure 16.3 `increaseKey(2, 8)`: the raised key can only be too large for its parent, so it rises; one swap restores the invariant.

Figure 16.3: 8 cannot harm its children (at most 5); it can only be too large for its parent 6. Result `[8, 5, 6, 4, 5, 3, 2, 1, 3, 1]`. *Book: Section 16.3.3 and Figure 16.3.*

`getMin`: Only the Leaves

$$\text{leaves} = \{\lfloor n/2 \rfloor, \lfloor n/2 \rfloor + 1, \ldots, n-1\}, \text{about } n/2 \text{ of them}. \tag{16.3}$$

Listing 16.2 `getMin` from `p1601_max_heap.cpp`: scan the leaves only.

```
    // 3. The minimum of a max-heap is a leaf; the leaves are heap[n/2 .. n-1].
    int getMin() const {
        if (heap.empty()) throw underflow_error("getMin on an empty heap");
        size_t n = heap.size();
        int minVal = heap[n / 2];
        for (size_t i = n / 2 + 1; i < n; i++)
            if (heap[i] < minVal) minVal = heap[i];
        return minVal;
    }
```

Equation (16.3): a node with a child is not the minimum, so Listing 16.2 looks at $\lceil n/2 \rceil$ elements: half a scan, still $\Theta(n)$. Book: Section 16.3.3.

The Minimum Is a Leaf

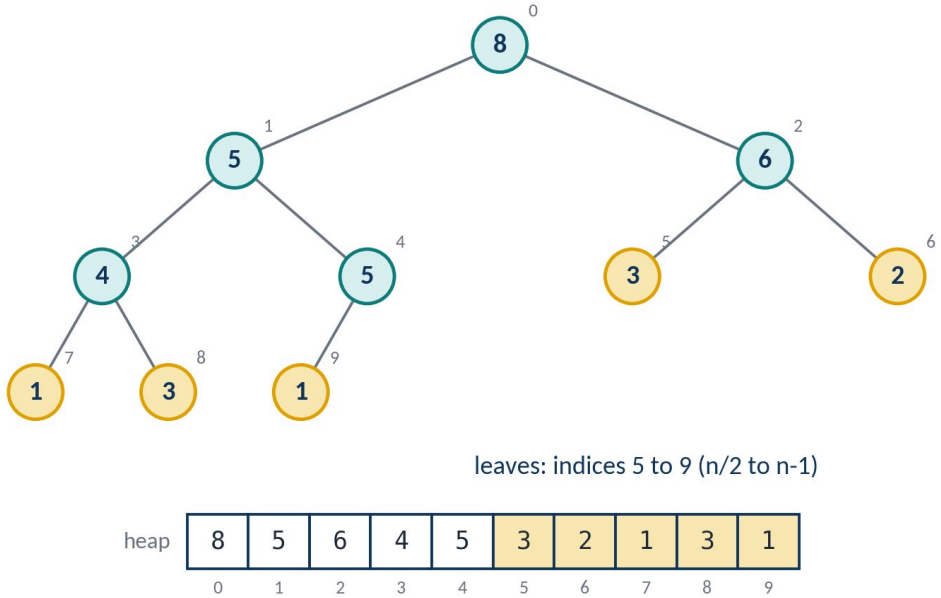


Figure 16.4 The minimum of a max-heap is always a leaf, and the leaves are the last half of the array: indices $\lfloor n/2 \rfloor$ to $n-1$.

Figure 16.4: for $n=10$ the leaves are indices 5 to 9, holding 3, 2, 1, 3, 1; the minimum is 1. Book: Section 16.3.3 and Figure 16.4.

`buildMaxHeap`, Bottom Up

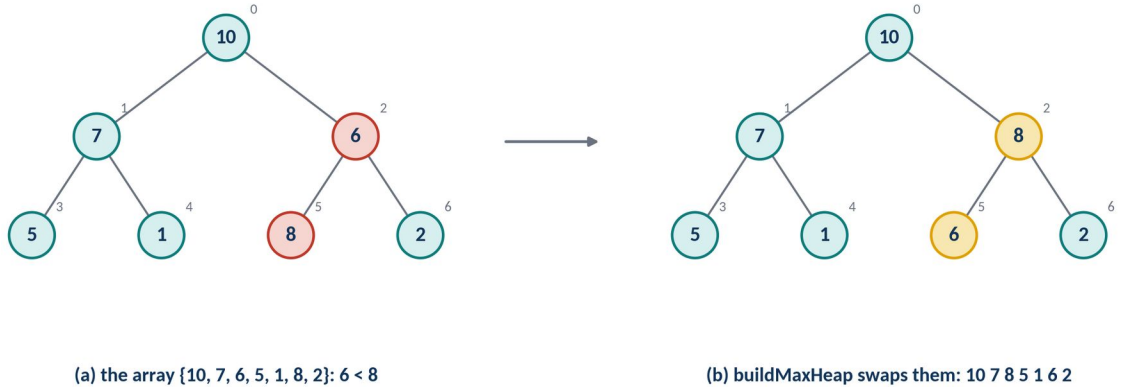


Figure 16.5 `buildMaxHeap` on {10, 7, 6, 5, 1, 8, 2}: only index 2 violates the invariant, and one swap repairs it.

Figure 16.5: `heapifyDown` on indices 2, 1, 0; only index 2 needs a swap: `[10, 7, 8, 5, 1, 6, 2]`. Book: Section 16.3.3 and Figure 16.5.

Why Building Is Linear, and the Check

$$\sum_{j=0}^{\lfloor \lg n \rfloor} \frac{n}{2^{j+1}}\, j = \frac{n}{2} \sum_{j \geq 0} \frac{j}{2^{j}} \leq \frac{n}{2} \cdot 2 = n. \tag{16.4}$$

Listing 16.3 `buildMaxHeap` and `isMaxHeap` from `p1601_max_heap.cpp`.

```
    // 4. Floyd's bottom-up construction: sift down every internal node, last first.
    void buildMaxHeap(const vector<int>& arr) {
        heap = arr;
        for (size_t i = heap.size() / 2; i-- > 0;) heapifyDown(i);
    }

    // 5. Every element except the root must be <= its parent.
    static bool isMaxHeap(const vector<int>& arr) {
        for (size_t i = 1; i < arr.size(); i++)
            if (arr[(i - 1) / 2] < arr[i]) return false;
        return true;
    }
```

Equation (16.4): most elements are near the bottom, where sinking is cheap. Listing 16.3: `i-- > 0` counts down without ever computing -1*; `isMaxHeap` loops over children from 1. Book: Sections 16.3.3–16.3.4.*

Wrong: the Last Element, and `size() - 2`

```
int getMinWrong(const vector<int>& heap) { return heap.back(); }       // WRONG
for (size_t i = 0; i <= (arr.size() - 2) / 2; i++)                    // WRONG for n < 2
```

On the driver's heap `getMinWrong` answers 1 by luck; on `[5, 3, 4]` it answers 4, but the minimum is 3.

For one element, `arr.size() - 2` is not -1 but about 1.8×10^{19}: AddressSanitizer reports *heap-buffer-overflow* in `isMaxHeapWrong`.

The last element is a leaf, but only one of about $n/2$ leaves. Loop over children, `i = 1; i < arr.size()`, whose bound cannot underflow. Book: Section 16.3.6.

Problem 16.2 — Putting Tickets in Travel Order

> **PROBLEM.** Given a list of tickets, find the itinerary in travel order, using hashing: a hash map `dataset` (*from* → *to*), a reverse map `reverseMap` (*to* → *from*); the key of `dataset` not found in `reverseMap` is the start. Input: Cianjur → Bandung, Badung → Denpasar, Gianyar → Cianjur, Denpasar → Gianyar.

The first ticket's departure city is **never an arrival city**; every other ticket leaves the city the previous one arrived at.

A pile *forms one chain* when the departures are all different, exactly one is not an arrival, and the walk from it uses every ticket.

Badung (Bali) and Bandung (West Java) are two different places; the one-letter difference is part of what makes the sample a good test. Book: Sections 16.4.1–16.4.2.

Worked Example: Tickets by Hand

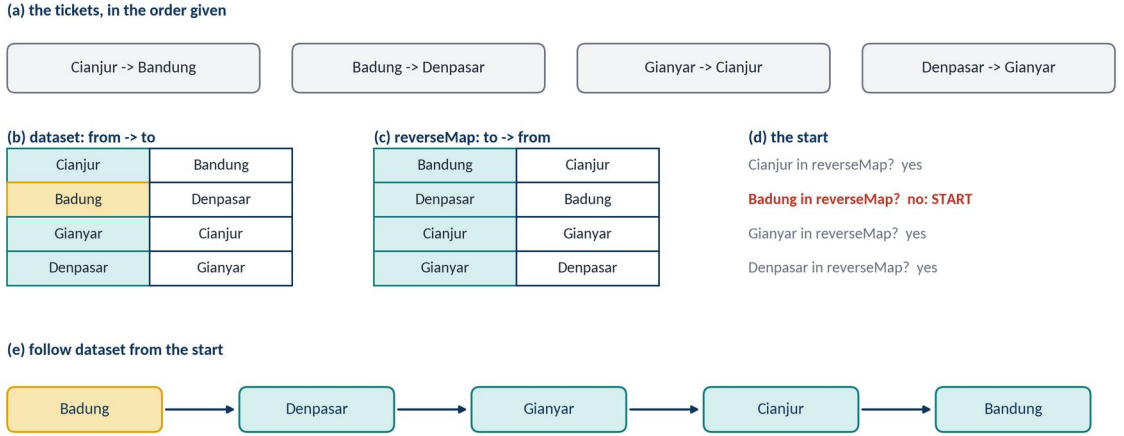


Figure 16.6 Problem 16.2 by hand: the tickets as given (a), the map from departure to arrival (b) and its reverse (c), the search for the one departure city that is never an arrival (d), and the walk from it (e).

Figure 16.6: Badung is the only departure that is never an arrival; four lookups print the trip. $O(n)$ on average, against $O(n^2)$ for searching the pile each time. Book: Section 16.4.3 and Figure 16.6.

Which Map Is a Hash Map?

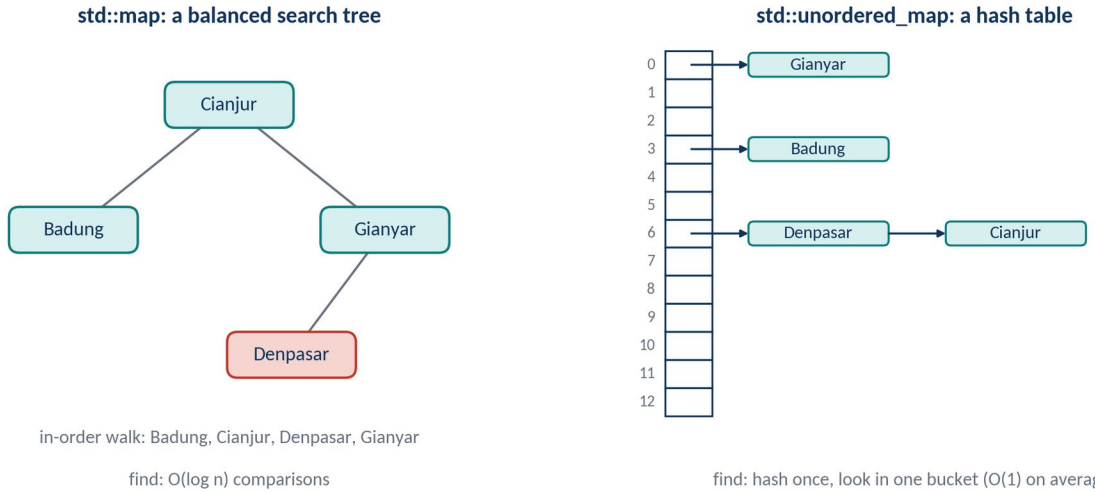


Figure 16.7 The same keys in `std::map`, a red-black tree kept in sorted order (the red node is a red node of the red-black scheme), and in `std::unordered_map`, a hash table with 13 buckets in which Cianjur and Denpasar collide.

Figure 16.7: `std::map` visits Badung, Cianjur, Denpasar, Gianyar in sorted order and needs 4 comparisons to find Gianyar; `std::unordered_map` hashes once and looks in one bucket. Book: Section 16.4.4 and Figure 16.7.

`findItinerary` in C++

Listing 16.4 `findItinerary` from `p1602_itinerary.cpp`: two hash maps, the start, and the walk.

```
    for (const Ticket& t : tickets) {
        if (!dataset.emplace(t.from, t.to).second)
            return "two tickets leave " + t.from;
        reverseMap[t.to] = t.from;

    for (const Ticket& t : tickets)
        if (reverseMap.find(t.from) == reverseMap.end()) starts.push_back(t.from);
    if (starts.empty()) return "no starting point (the tickets form a loop)";

    string city = starts[0];
    auto it = dataset.find(city);
    while (it != dataset.end() && route.size() < tickets.size()) {
        route.push_back({it->first, it->second});
        it = dataset.find(it->second);
    }
    if (route.size() != tickets.size())
        return "the tickets do not form one chain";
```

Listing 16.4: the start is searched in input order, never in the table's unspecified order; the walk stops after n tickets. Book: Section 16.4.5.

Wrong: Start From the First Ticket

```
  string city = tickets[0].first;      // WRONG: "the first ticket is the first leg"
```

```
Cianjur -> Bandung
```

The first ticket in the pile is the *last* leg; from Bandung there is no ticket, and three quarters of the trip is lost without a word.

A second slip: a comma counter declared `int a;` and never set to 0; Valgrind reports "Conditional jump or move depends on uninitialised value(s)". Book: Section 16.4.7.

Problem 16.3 — The Fewest Hops Between Branches

PROBLEM. Roads: Pusat–Gubeng, Pusat–Darmo, Pusat–Kenjeran, Gubeng–Sukolilo, Gubeng–Rungkut, Darmo–Wonokromo, Wonokromo–Rungkut, Kenjeran–Sukolilo; Benowo has none. Every road is one hop. Find the fewest hops from Pusat to every branch, and print three routes.

DEFINITION. The *BFS invariant*: the queue holds vertices in nondecreasing order of `dist`, first and last differing by at most 1. So when v is discovered from u, `dist[v] = dist[u] + 1` is final.

Set `dist[v]` when v is first seen, not when it leaves the queue; -1 means unseen, so one array serves as distances and visited marks. Book: Section 16.5.1.

Rings of Equal Distance

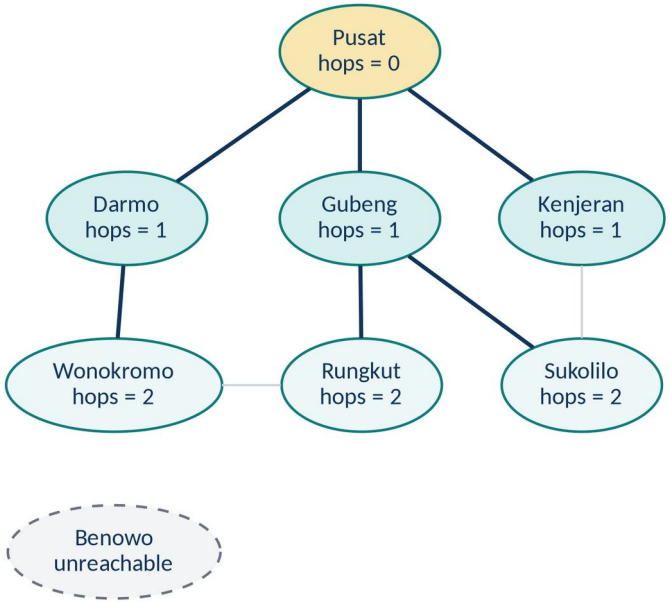


Figure 16.8 The branch map searched breadth-first from Pusat: vertices in rings of equal distance, the BFS tree in dark lines, the two unused roads pale, and Benowo unreachable.

Figure 16.8: dark edges form the BFS tree; the route to Rungkut, read back through the parents, is Pusat → Gubeng → Rungkut. Book: Section 16.5.1 and Figure 16.8.

`bfs` in C++

Listing 16.5 `bfs` from `p1603_bfs_hops.cpp`: mark before queueing; first sight is final.

```
    dist.assign(adj.size(), -1);
    parent.assign(adj.size(), -1);
    queue<int> q;
    dist[s] = 0;                        // mark s BEFORE it waits in the queue
    q.push(s);
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        order.push_back(u);
        for (int v : adj[u]) {
            if (dist[v] == -1) {        // first sight of v: its distance is final
                dist[v] = dist[u] + 1;
                parent[v] = u;
                q.push(v);
            }
        }
    }
```

$$T_{\text{BFS}} = O(|V| + |E|). \tag{16.5}$$

Listing 16.5 costs Equation (16.5). Python must use `collections.deque`, never `list.pop(0)`. Book: Section 16.5.1.

Wrong: DFS Depth as a Distance

```
void dfs(int u, int d) {                  // WRONG: depth in a DFS is not a distance
    dist[u] = d;
    for (int v : adj[u]) if (dist[v] == -1) dfs(v, d + 1);
}
```

```
Pusat 0
Gubeng 1
Darmo 4
Kenjeran 3
```

Darmo is one road from Pusat, but DFS dived through Gubeng, Sukolilo, Kenjeran, Rungkut, Wonokromo first and reached it last.

DFS answers “can I get there?”; only the ring-by-ring order of a queue answers “how few hops?”. Book: Section 16.5.1.

Problem 16.4 — An Order to Read the Topics

PROBLEM. The eleven topics of the book depend on each other (arrays before searching, sorting, stacks and pointers; …). Print an order in which every topic comes after all it needs. Then add “graphs before pointers” and report what happens.

DEFINITION. The *Kahn invariant*: every topic in the order has all its prerequisites earlier, and `indeg[v]` counts the prerequisites of v not yet placed. If the queue empties with topics missing, there is a directed cycle.

Each “read u before v” is an arc; an order with every arc pointing forward is a topological order, which exists exactly for a DAG. Book: Section 16.5.2.

Eleven Topics, Sixteen Arcs

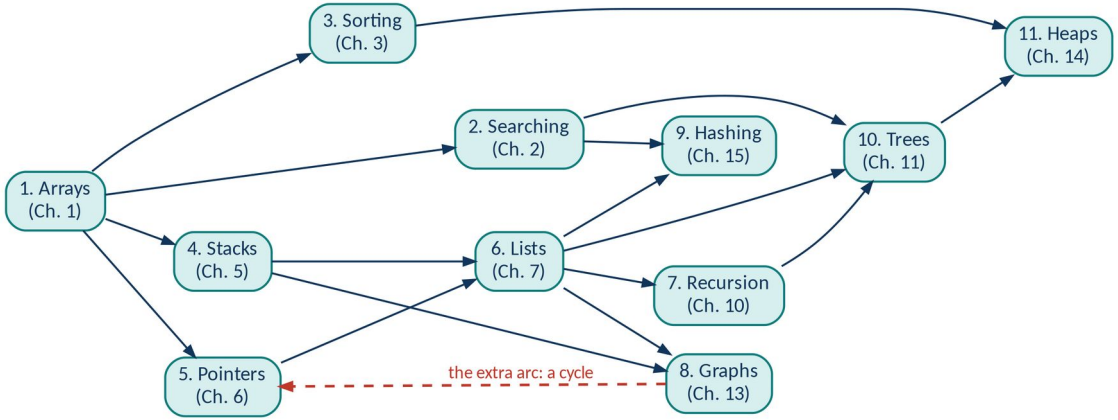


Figure 16.9 The eleven topics and the sixteen "read before" arcs, each topic numbered by its place in the order Kahn's algorithm finds; the dashed red arc "graphs before pointers" closes the cycle Pointers, Lists, Graphs, Pointers.

Figure 16.9: Graphs and Hashing come before Trees: valid, since no arc forbids it. With the red arc, only Arrays, Searching, Sorting, Stacks are placed. Book: Section 16.5.2 and Figure 16.9.

`kahn` in C++

Listing 16.6 `kahn` from `p1604_reading_order.cpp`: in-degrees, a queue of ready topics, and `--indeg[v] == 0`.

```
vector<int> kahn(const vector<vector<int>>& adj) {
    int n = adj.size();
    vector<int> indeg(n, 0);
    for (int u = 0; u < n; u++)
        for (int v : adj[u]) indeg[v]++;
    queue<int> ready;                          // topics with no unread prerequisite
    for (int u = 0; u < n; u++)
        if (indeg[u] == 0) ready.push(u);
    vector<int> order;
    while (!ready.empty()) {
        int u = ready.front();
        ready.pop();
        order.push_back(u);
        for (int v : adj[u])
            if (--indeg[v] == 0) ready.push(v);    // v's last prerequisite is read
    }
    return order;
}
```

Listing 16.6 costs $O(|V|+|E|)$, as BFS. It returns the order; the caller compares `order.size()` with n. Book: Section 16.5.2.

Wrong: No Count After the Loop

```
Reading order: Arrays Searching Sorting Stacks
```

With the cycle, the program prints four topics and no warning; the other seven silently vanish.

On every input without a cycle it is perfect, which is why the omission survives testing on the sample.

Step 5 of the method, which lists “a cycle” among the edge cases of a graph, is what catches it. Book: Section 16.5.2.

Problem 16.5 — The Quickest Route for the Book Van

PROBLEM. The roads now have minutes: Pusat–Gubeng 5, Pusat–Darmo 9, Pusat–Kenjeran 14, Gubeng–Sukolilo 12, Gubeng–Rungkut 6, Darmo–Wonokromo 4, Wonokromo–Rungkut 3, Kenjeran–Sukolilo 4, and a direct Pusat–Rungkut of 20. Find the quickest times, using a priority queue.

DEFINITION. The *Dijkstra invariant*: every branch marked done has its quickest time; the branch taken out of the heap with the smallest tentative time can be declared done, because lengths are never negative.

Lazy deletion: when `dist[v]` *drops, push a new entry; an entry for a branch already done is stale and skipped. Book: Section 16.5.3.*

Worked Example: The Trace

```
take Pusat (0)
  Gubeng: inf -> 5
  Darmo: inf -> 9
  Kenjeran: inf -> 14
  Rungkut: inf -> 20
take Gubeng (5)
  Sukolilo: inf -> 17
  Rungkut: 20 -> 11
take Darmo (9)
  Wonokromo: inf -> 13
take Rungkut (11)
take Wonokromo (13)
take Kenjeran (14)
take Sukolilo (17)
skip Rungkut (20): an old, stale entry
```

Gubeng at 5 is done: no route through Darmo (9) or Kenjeran (14) can beat it. From Gubeng, Rungkut drops from 20 to 11; the old entry stays and is skipped at the end. Book: Section 16.5.3.

The Shortest-Path Tree

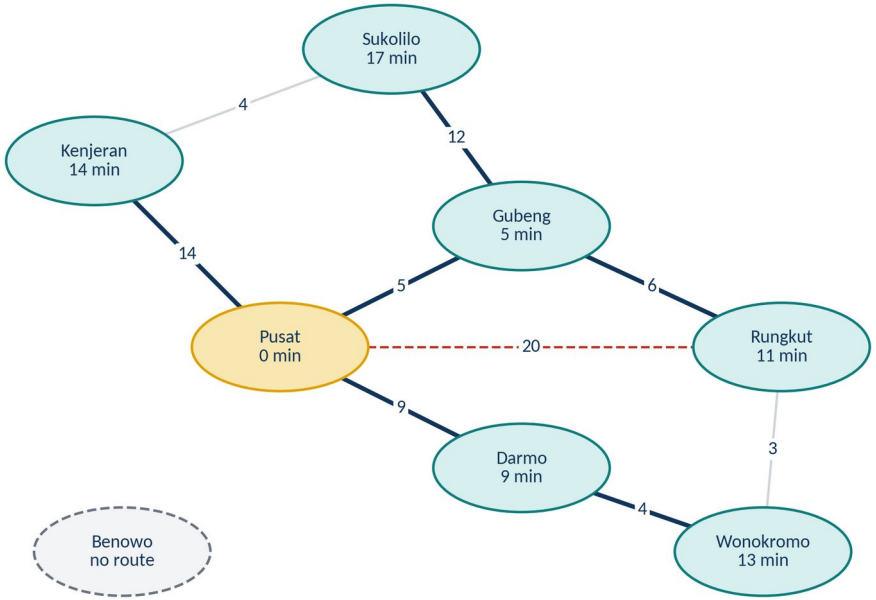


Figure 16.10 Quickest times from Pusat. Dark roads form the shortest-path tree; the dashed red direct road Pusat–Rungkut (20 minutes) loses to the detour through Gubeng (11 minutes).

Figure 16.10: the direct road (20) loses to the detour through Gubeng (11). Book: Section 16.5.3 and Figure 16.10.

`dijkstra` in C++

Listing 16.7 The heart of `dijkstra` from `p1605_dijkstra.cpp` (trace printing left out): a min-heap of (time, branch) pairs with lazy deletion.

```
    while (!heap.empty()) {
        auto [d, u] = heap.top();
        heap.pop();
        if (done[u]) {                                  // an old, larger entry
            cout << "skip " << NAME[u] << " (" << d << "): an old, stale entry\n";
            continue;
        }
        done[u] = true;                                 // d is now final for u
        cout << "take " << NAME[u] << " (" << d << ")\n";
        for (const Arc& a : adj[u]) {
            if (!done[a.to] && d + a.minutes < dist[a.to]) {

                dist[a.to] = d + a.minutes;
                parent[a.to] = u;
                heap.push({dist[a.to], a.to});
```

$$T_{\text{Dijkstra}} = O((|V|+|E|)\lg|V|). \tag{16.6}$$

Listing 16.7, cost Equation (16.6): `greater<Entry>` *turns the C++ max-heap into a min-heap (Table 16.3);* `if (done[u])` *is the stale-entry test. Book: Section 16.5.3.*

Wrong: Final on First Sight

```
  if (!seen[v]) { seen[v] = true; dist[v] = d + w; heap.push({dist[v], v}); }   // WRONG
```

```
Pusat 0
Gubeng 5
Darmo 9
Kenjeran 14
Rungkut 20
```

Rungkut is sealed at 20 before Gubeng, which offers 11, has even been taken.

In BFS first sight is final because every edge counts 1; with lengths a time is final only when its branch leaves the heap as the smallest. Book: Section 16.5.3.

Problem 16.6 — The k Largest with a Heap of Size k

PROBLEM. (a) Eight books were borrowed 12, 45, 7, 30, 18, 3, 26 and 40 times. Find the three most borrowed with a heap that never holds more than three. (b) Find the five most-rated books of the ten-thousand-book catalog, and check against sorting.

DEFINITION. The *top-k invariant*: after i values, the heap holds the $\min(i, k)$ largest of them, and its root is the smallest it holds.

*Ask of every value: is it better than the worst I keep? The worst must be found at once, so the heap is a **min**-heap. Book: Section 16.6.1.*

Worked Example: Three Most Borrowed

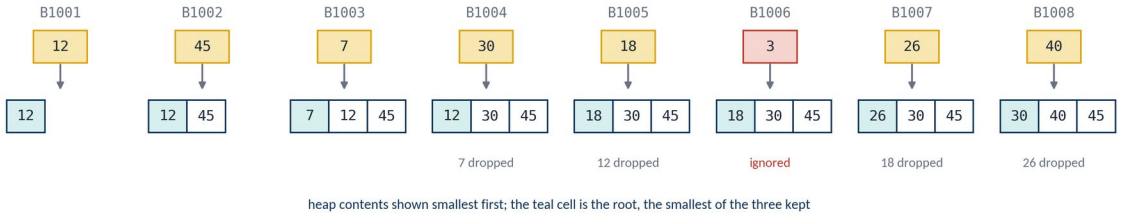


Figure 16.11 The three most borrowed books found with a min-heap of size 3. Each new count is compared with the root, the smallest of the three kept; if it is larger, the root is dropped.

Figure 16.11: B1006 (3) does not beat the root 18 and costs one comparison. Answer: B1002 (45), B1008 (40), B1004 (30). Book: Section 16.6.1 and Figure 16.11.

`offer` in C++

Listing 16.8 `TopK::offer` from `p1606_top_k.cpp`: a min-heap of size k in a `vector`, with `push_heap` and `pop_heap`.

```
bool offer(const Item& x) {
    if (heap.size() < k) {                    // not full yet: always keep
        heap.push_back(x);
        push_heap(heap.begin(), heap.end(), greater<Item>());
        return true;
    }
    if (x <= heap.front()) return false;      // not above the smallest kept
    pop_heap(heap.begin(), heap.end(), greater<Item>());   // drop the smallest
    heap.back() = x;
    push_heap(heap.begin(), heap.end(), greater<Item>());
    replaced++;
    return true;
}
```

$$T_{\text{top-}k} = O(n \lg k), \text{extra memory } O(k). \tag{16.7}$$

Listing 16.8, cost Equation (16.7): `x <= heap.front()` *rejects in* $O(1)$*; a larger value replaces the root in* $O(\lg k)$*. Python's* `heapq.heapreplace` *does the same in one sift-down. Book: Section 16.6.1.*

How Many Replacements?

$$\sum_{i=k+1}^{n} \frac{k}{i} \approx k \ln \frac{n}{k}, \tag{16.8}$$

```
Changes to the heap after it filled up:
  0 in file order (most-rated first), 9 in identifier order,
  9995 in increasing order of ratings.
  1. 4942365  The Hunger Games (The Hunger Games, #1)
  2. 4800065  Harry Potter and the Sorcerer's Stone (Harry Potter, #1)
Same five as sorting all 10000 books: yes
```

Equation (16.8): about $5 \ln 2000 \approx 38$ *in random order; the best order gives 0, the worst gives every one of the other 9,995. Book: Section 16.6.1.*

Wrong: a Max-Heap

```
  priority_queue<int> heap;            // WRONG: a MAX-heap; top() is the largest
  heap.push(c); if (heap.size() > 3) heap.pop();
```

```
Top 3: 12 7 3
```

Every `pop` removes the largest seen so far, exactly the one we wanted to keep: the result is the three *smallest*.

The same code is right with Java's `PriorityQueue` or Python's `heapq`, which are min-heaps: the first dangerous row of Table 16.3. Book: Section 16.6.1.

Problem 16.7 — The Median So Far, with Two Heaps

PROBLEM. Visitors waited 5, 15, 1, 3, 8, 7, 9, 10, 20 and 2 minutes. After each, print the median so far. Each update must cost $O(\lg n)$, not a sort.

DEFINITION. The *two-heap invariant*: (i) every value in the max-heap `low` is at most every value in the min-heap `high`; (ii) `low.size()` equals `high.size()` or `high.size() + 1`. The median is `low.top()`, or the average of the two tops.

Place the value by comparing with `low.top()`; if the sizes break (ii), move one top across. Moving a top keeps (i). Book: Section 16.6.2.

The Two Heaps After Ten Values

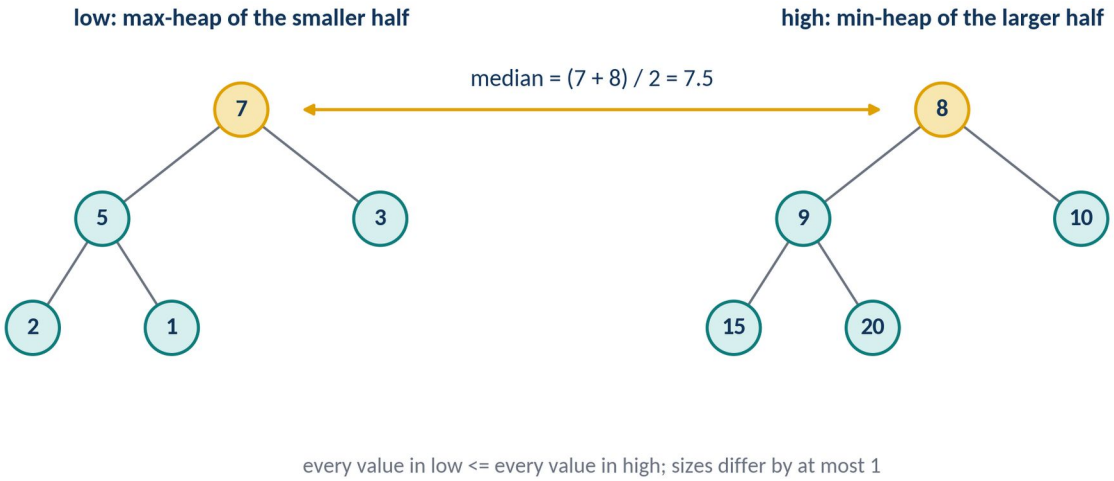


Figure 16.12 The two heaps after all ten waiting times. The roots are the two middle values of the sorted list, so the median is read off in constant time.

Figure 16.12: the roots 7 and 8 are the middle of 1, 2, 3, 5, 7, 8, 9, 10, 15, 20; the median is 7.5, read in $O(1)$. Book: Section 16.6.2 and Figure 16.12.

`RunningMedian` in C++

Listing 16.9 `add` and `median` from `p1607_running_median.cpp`: a max-heap for the lower half, a min-heap for the upper half.

```cpp
    void add(int x) {
        if (low.empty() || x <= low.top()) low.push(x);
        else high.push(x);
        if (low.size() > high.size() + 1) {              // low too big: move top
            high.push(low.top());
            low.pop();
        } else if (high.size() > low.size()) {           // high too big: move top
            low.push(high.top());
            high.pop();
        }
    }
    double median() const {
        if (low.size() > high.size()) return low.top();
        return (low.top() + high.top()) / 2.0;
    }
```

Listing 16.9: here the C++ default max-heap is exactly right for `low`; Python stores `low` negated. `2.0` makes the average 7.5, not 7. Book: Section 16.6.2.

Wrong: No Rebalancing

```
add 5  median 5
add 15  median 10
add 1  median 5
add 3  median 10
add 8  median 5
```

After 5, 15, 1, 3 the halves are {5, 3, 1} and {15}: part (i) holds, part (ii) does not, and the fourth median is 10 instead of 4.

The median is a statement about positions; only the size condition ties the roots to the middle positions. Book: Section 16.6.2.

Problem 16.8 — Separate Chaining by Hand

PROBLEM. Store eight books, keyed by title, in a chained hash table that starts with 5 buckets and grows to $2m+1$ when $n>m$. Hash: $h=(h\cdot 31+c)\bmod m$. Look up two present titles and one absent; remove one title twice.

$$\alpha=\frac{n}{m}, \tag{16.9}$$

$$T_{\text{chain}}=\Theta(1+\alpha). \tag{16.10}$$

Equation (16.9) is the average chain length; by Equation (16.10), keeping $\alpha\le 1$ keeps every operation $O(1)$ on average. The chaining invariant: key k is in chain $h(k)$ exactly once. Book: Section 16.7.1.

Before and After the Rehash

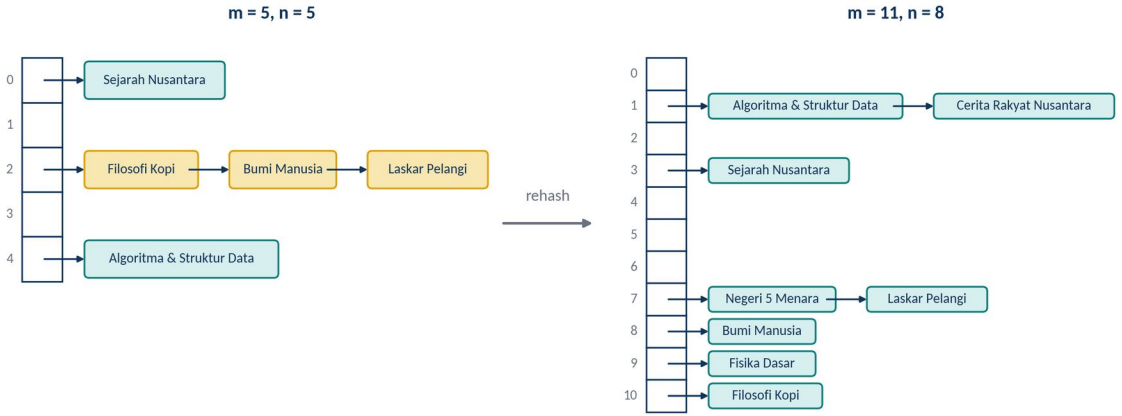


Figure 16.13 The chained table with 5 buckets (three titles collide in bucket 2) and after rehashing to 11 buckets, where no chain is longer than two.

Figure 16.13: with 5 buckets three titles share bucket 2; the sixth title makes $6>5$*, and every node is relinked into 11 buckets. Book: Section 16.7.1 and Figure 16.13.*

`hashOf` and `remove` in C++

Listing 16.10 `hashOf` and `remove` from `p1608_chaining.cpp`: unsigned arithmetic reduced at every step, and a pointer to the link.

```
    // Polynomial hash, reduced modulo m at every step so it never overflows.
    static size_t hashOf(const string& key, size_t m) {
        size_t h = 0;
        for (unsigned char c : key) h = (h * 31 + c) % m;
        return h;
    }

    bool remove(const string& key) {
        Node** link = &bucket[indexOf(key)];              // the pointer to change
        while (*link != nullptr && (*link)->key != key) link = &(*link)->next;
        if (*link == nullptr) return false;
        Node* dead = *link;
        *link = dead->next;                               // unlink, THEN free
        delete dead;
        n--;
        return true;
    }
```

Listing 16.10: `Node** link` *points first at the bucket's head, then at each* `next`*;* `*link = dead->next` *unlinks without a special case for the first node. Valgrind: no leak. Book: Section 16.7.1.*

Wrong: the Hash in a Signed `int`

```
int h = 0;
for (char c : key) h = h * 31 + c;   // WRONG: overflows for keys of 7+ letters
return h % m;                        // ...and a negative h gives a negative index
```

```
Laskar Pelangi -> bucket -1
Segmentation fault
```

UBSan: *signed integer overflow: 73191252 * 31 cannot be represented in type 'int'*; the negative remainder gives index −1.

Use unsigned arithmetic and reduce at every step; in Java, `Math.floorMod(h, m)`. Python never overflows, but the value grows to hundreds of digits. Book: Section 16.7.1.

Problem 16.9 — Open Addressing, Deleting Safely

PROBLEM. Insert 10, 22, 31, 4, 15, 28, 17, 88, 59 into 11 slots with $h(k) = k \bmod 11$ and linear probing. Then remove 15, search for 59 and 26, and insert 37.

DEFINITION. The *probing invariant*: if k is in slot j, every slot from $h(k)$ up to j is occupied or a *tombstone*. A search skips tombstones and stops only at an empty slot.

$$S \approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right), U \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right). \tag{16.11}$$

Equation (16.11): at $\alpha = 9/11$, about 3.3 and 15.6 probes. Emptying a slot can cut the probe path of a key pushed past it. Book: Section 16.7.2.

Worked Example: a Tombstone in Slot 5

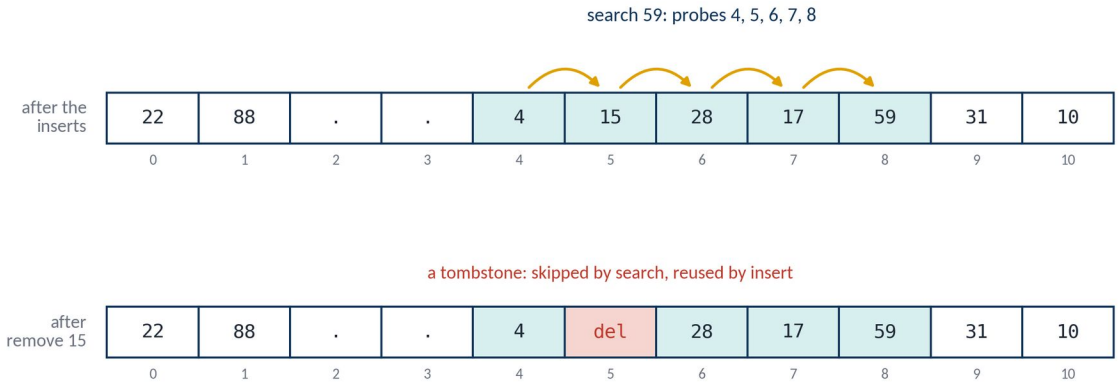


Figure 16.14 Linear probing with $m = 11$: the probe path of 59 runs through slots 4 to 8. Removing 15 leaves a tombstone in slot 5, which a search skips and an insertion reuses.

Figure 16.14: 59 probes 4, 5, 6, 7, 8. After removing 15, search 59 skips the tombstone; 37 reuses slot 5 only after reaching the empty slot 2. Book: Section 16.7.2 and Figure 16.14.

`search` and `remove` in C++

Listing 16.11 `search` and `remove` from `p1609_linear_probing.cpp`: EMPTY ends a search, DELETED does not.

```
    // Returns the slot of k, or -1. A tombstone does NOT stop the search.
    int search(int k, string& trace) const {
        for (int j = 0; j < m; j++) {
            int i = (h(k) + j) % m;
            trace += " " + to_string(i);
            if (state[i] == EMPTY) return -1;            // k would have been here
            if (state[i] == FULL && key[i] == k) return i;
        }
        return -1;                                       // looked at every slot
    }

    bool remove(int k) {
        string ignored;
        int i = search(k, ignored);
        if (i == -1) return false;
        state[i] = DELETED;                      // NOT EMPTY: keep the runs whole
        return true;
    }
```

Listing 16.11. `insert` *remembers the first free slot but continues to an EMPTY slot to prove the key absent. Book: Section 16.7.2.*

Wrong: Deleting by Emptying the Slot

```
before: search 59 -> slot 8
after removing 15: search 59 -> slot -1
```

59 is still in slot 8, but its search meets the emptied slot 5 and stops: the table *loses* a key it holds.

A book on the shelf that the computer swears is not there. Chaining cannot make this mistake.

Tombstones make every sequence of operations safe, not just one; a rehash sweeps them away. Book: Section 16.7.2.

Choosing the Right Data Structure

Table 16.4 Choosing a data structure by the operation the task performs most often (a shortened form of the book's table).

The task mostly needs to...	Choose	Cost	Ch.
read or write item number i	array / `vector`	$O(1)$	1
find by key, list in order, rare changes	sorted array + binary search	$O(\lg n)$	2, 3
add and remove at the most recent end	stack	$O(1)$	5
serve in order of arrival	queue	$O(1)$	5, 9
insert or delete at a node you hold	linked list	$O(1)$	7, 9
find by key, plus order or ranges, frequent changes	balanced BST	$O(\lg n)$	11
repeatedly take the largest or smallest	heap / priority queue	$O(\lg n)$	14
find, insert or delete by key, nothing else	hash table	$O(1)$ average	15
model connections	graph with adjacency lists	$O(\lvert V\rvert+\lvert E\rvert)$	13

Table 16.4: the whole book on one page. Name the dominant operation first. Book: Section 16.8.1.

The Decision as a Tree

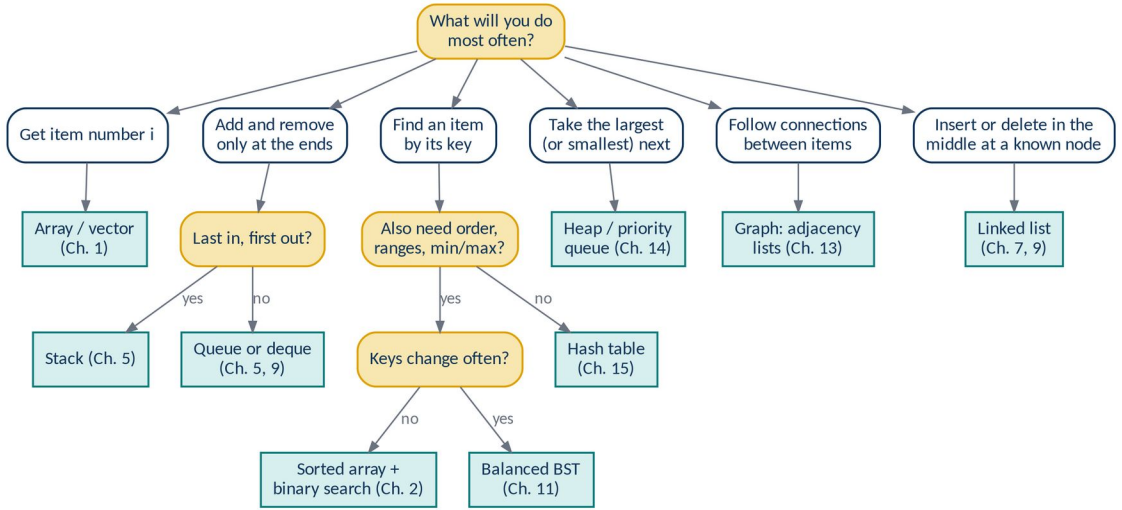


Figure 16.15 Choosing a structure: begin with the operation the task performs most often, then answer at most two follow-up questions. The chapter that teaches each structure is in brackets.

Figure 16.15: "find by key" leads to a hash table only if you never need order; with "smallest", "next" or "between", choose a search tree or a sorted array. Book: Section 16.8.1 and Figure 16.15.

The Experiment: Six Structures, One Catalog

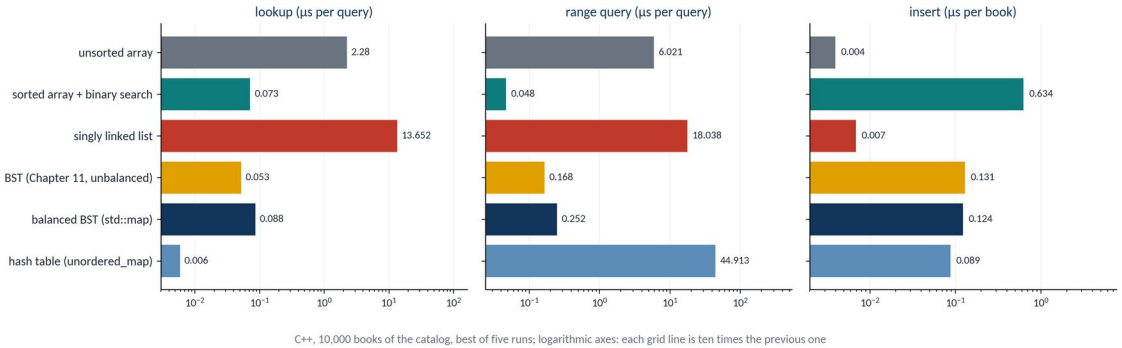


Figure 16.16 Six structures on the Pustaka Ceria catalog (C++, microseconds per operation, best of five runs, logarithmic axes). No structure wins every column.

Figure 16.16: build, 2,000 lookups (half absent), 200 range queries, 1,000 inserts on 10,000 real identifiers; every structure must give the same answers. Book: Sections 16.8.2–16.8.3 and Figure 16.16.

Reading the Numbers

Table 16.5 Microseconds per lookup and per range query on the 10,000-book catalog, from the three runs above. The best structure for lookup is among the worst for ranges.

Structure	Lookup C++	Lookup Java	Lookup Python	Range C++	Range Java	Range Python
unsorted array	2.280	1.546	70.476	6.021	9.619	190.958
sorted array + binary search	0.073	0.206	0.336	0.048	0.182	0.451
singly linked list	13.652	19.141	139.348	18.038	29.328	503.491
BST (Chapter 11, unbalanced)	0.053	0.158	0.556	0.168	0.333	3.624
balanced BST (library)	0.088	0.269	—	0.252	1.763	—
hash table (library)	0.006	0.169	0.034	44.913	113.180	255.484

$$|V|^2 = 5841^2 = 34,117,281 \text{ cells against } |V| + 2|E| = 19,835 \text{ list entries}, \tag{16.12}$$

Table 16.5: the hash table wins lookup and loses ranges; the sorted array is best for ranges and slowest to insert into. Equation (16.12): the people graph (5,841 people) belongs in adjacency lists. Book: Section 16.8.3.

The Problems Compared

Table 16.6 The nine problems: the structure each uses, its cost, and the input on which its tempting wrong solution fails.

Problem	Structure	Time	The wrong solution fails on
16.1	array max-heap	$O(\lg n)$; build $O(n)$	a last element that is not the minimum
16.2	two hash maps	$O(n)$ average	a first ticket that is not the first leg
16.3	lists + queue	$O(\|V\|+\|E\|)$	DFS reaching a vertex the long way
16.4	lists + queue	$O(\|V\|+\|E\|)$	a directed cycle
16.5	lists + min-heap	$O((\|V\|+\|E\|)\lg\|V\|)$	a direct road longer than a detour
16.6	min-heap of size k	$O(n \lg k)$	any input, with a max-heap
16.7	max-heap + min-heap	$O(\lg n)$ per value	the fourth value, without rebalancing
16.8	chained hash table	$O(1+\alpha)$ average	a key that overflows an `int`
16.9	linear probing	$O(1/(1-\alpha)^2)$ average	a path through a deleted slot

Table 16.6: most tempting wrong solutions pass the example they were written for. Three cost families: one path down a tree, one pass, one computed jump. Book: Section 16.9.

Profitina in Practice

A website is a graph: pages are vertices, links are arcs; orphan pages are unreachable, click depth is a BFS distance.

The seen set is a hash table: URLs are canonicalized before hashing, so one page is not stored three times.

Rescans are a priority queue keyed by due time; **top lists** are heaps of size k.

Profitina is an AI-visibility (GEO) platform; its server is Python, so these appear as `dict`, `set`, `deque` *and* `heapq`*. We describe the shape of each problem, not how the product is built. Book: Section 16.10.*

Common Mistakes

Heaps: the wrong end of a priority queue; a max-heap for the k largest; not saving the root; repairing in the wrong direction; `size() - 2` underflow; forgetting to rebalance two heaps.

Graphs: DFS depth as a distance; marking on leaving the queue; a topological sort without counting; a Dijkstra distance fixed on first sight.

Hash tables: calling `std::map` a hash map; relying on iteration order; a hash in a signed `int`; deleting by emptying a slot; letting the table fill up.

And choosing a structure before naming the operation: name it, consult Table 16.4, then measure. Book: Section 16.11.

Summary: Five Things to Remember

1. Nine steps: restate, choose, draw, invariant, edge cases, safe order, cost, bad input, test against a slow twin.

2. Every structure promises an invariant; every operation must leave it true.

3. Heaps: height $\lfloor \lg n \rfloor$; minimum among the leaves; build in $O(n)$; the k largest need a *min*-heap; a median needs two.

4. Graphs: BFS for fewest hops, Kahn for an order or a cycle, Dijkstra with a heap for weighted routes. Hash tables: keep α small, no overflow, tombstones.

5. Choose by the dominant operation, then measure.

Book: Section 16.12.

Where to Go From Here

This is the last chapter of the book. Arrays, lists, stacks and queues, recursion, trees, graphs, heaps and hash tables are now one toolbox.

Work Exercises 16.1, 16.3, 16.10 and 16.12, and build the Profitina Laundry service desk of Exercise 16.14: a hash map of orders and a priority queue of promised times.

Faced with a real task: name the operation, choose from Table 16.4, and measure before you trust either.

Solutions, with programs in all three languages, are in the book. Book: Sections 16.13 and 16.14.